

IBM Copyright Permission #22527

Reprint Courtesy of International Business Machines Corporation, © 1994 International Business Machines Corporation'

INTERNATIONAL BUSINESS MACHINES CORPORATION (IBM) ARMONK, NEW YORK 10504

PERMISSION TO REPRINT/POST IBM COPYRIGHTED PUBLICATIONS

The material owned by IBM must be accompanied by the following credit line: “**Reprint Courtesy of International Business Machines Corporation, © [Year] International Business Machines Corporation**”. The credit line normally should appear on the page where the posting appears, either under the title or as a footnote. If the foregoing is inconvenient, the credit line may be placed in a conveniently viewable manner with suitable reference to the places where the material appears.

It is the understanding of **International Business Machines Corporation** that the purpose for which its material is being reproduced is accurate and true as stated in the original request.

Permission to quote from, transmit electronically or reprint/post IBM material is limited to the purpose and quantities originally requested and must not be construed as a blanket license to use the material for other purposes or to reproduce other IBM copyrighted material.

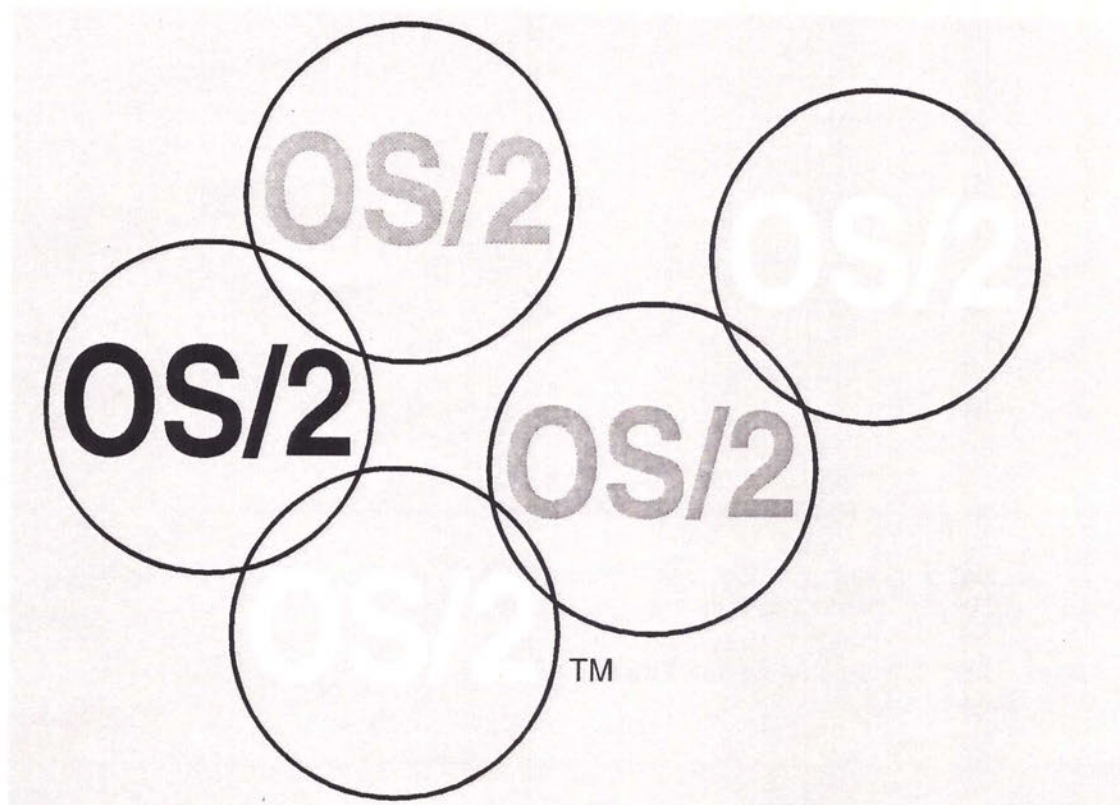
IBM reserves the right to withdraw permission to reproduce copyrighted material whenever, in its discretion, it feels that the privilege of reproducing its material is being used in a way detrimental to its interest or the above instructions are not being followed properly to protect its copyright.

No permission is granted to use trademarks of **International Business Machines Corporation** and its affiliates apart from the incidental appearance of such trademarks in the titles, text, and illustrations of the named publications. Any proposed use of trademarks apart from such incidental appearance requires separate approval in writing and ordinarily cannot be given. The use of any IBM trademark should not be of a manner which might cause confusion of origin or appear to endorse non-IBM products.

THIS PERMISSION IS PROVIDED WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

INTERNATIONAL BUSINESS MACHINES CORPORATION

Dated: April 22, 2014



**OS/2 2.1 for Software Developers
Supplemental Material
CN17400C/N1740**

Revision 1.04
September 13, 1994

Course materials may not be reproduced in whole or in part without the prior written permission of IBM Corporation.

© Copyright IBM Corporation (1991, 1994)
All Rights Reserved.

| | | |
|---------|--------------|--------------------------------|
| CONTACT | ORGANIZATION | Education and Training |
| | NODE(USERID) | DALVM1(ROHR) |
| | Internet | rohr@dalvm1.ibm.com |
| | NAME | Bob Rohr |
| | TELEPHONE | 914-742-5653 Tie/line 770-5653 |

Table of Contents

Appendix C - IBM C Set/2 Compiler Issues

Appendix D - OS/2 2.1 Desktop

Appendix DL - Dynamic Link Libraries

Appendix E - OS/2 2.1 Exception Handling

Appendix EDIT - PC DOS E.EXE and OS/2 EPM.EXE

Appendix L - OS/2 2.1 Extra Lab Projects

Appendix M - OS/2 2.1 Memory Management API

Appendix S - IBM C Set Compiler Signals

Appendix STAC - Using Stacker for OS/2 and Dos

Appendix T - IBM C Set Compiler Thunking

The following diskettes may be available to you:

1. N1740 Class Diskette
2. UPDATE.ZIP - Extra .INF files
3. WTDOS25C/N1746 Advanced OS/2 2.0 Class Diskette
4. HP PCL 5 for Advanced OS/2 2.0 Labs, for Supplement

Reading Assignments:

- Day 1: Read Appendix D, EDIT
- Day 2: Read Appendix C, M
- Day 3: Read Appendix DLL
- Day 4: Read Appendix E, S

NOTES:

Copy to floppies by doing:

```
XCOPY \OS20LABS\DISK1\*. * A: /S /V
XCOPY \OS20LABS\DISK2\*. * A: /S /V
XCOPY \OS20LABS\DISK3\*. * A: /S /V
XCOPY \OS20LABS\DISK4\*. * A: /S /V
```

The diskettes are stored in \OS20LABS\DISK1, etc. You are free to copy these to floppies. There is a read.me file in each directory. In general (except for #4) you can install the software simply by doing PKUNZIP2 -o -d a:*.zip.

Disk1 contains the workshop help, the text supplement, the review questions, the lab projects, the lab solutions, and the example programs.

The homework reading should prepare you better for the lecture material. Do as little or as much of the reading as you find comfortable. In general, reading the material ahead of time makes the lectures go a bit easier.

The lab disk (disk #3) and the lab text (disk #4) from the Advanced OS/2 class are provided for your use in reviewing the material. Disk #4 must be printed by copying the files to an HP Laserjet 3 or 4, or any printer supporting PCL5. Use COPY /B filename prn.

OS/2 2.1 for Software Developers

Appendix C

C Set/2 Compiler Issues

Some material in this chapter
courtesy IBM Toronto Lab

by Charles R. Chernack

Table of Contents

| | |
|---|------|
| Appendix C - IBM C Set/2 Compiler Issues | C-1 |
| IBM C Set Runtime Libraries (not including C++) . . . | C-3 |
| Typical Makefile | C-6 |
| IBM C Set Compiler Options | C-6 |
| Output File Management Options (/Fa, etc) . . . | C-6 |
| Listing Options (expand macros, etc) | C-7 |
| Lint Like Options (/Kb, etc) | C-8 |
| Debugging Options (/Ti, etc) | C-9 |
| Source Code Options (/Ss, etc) | C-10 |
| Code Generation Options (/Gd /Gm /Rn, etc.) . . | C-11 |
| Other Compiler Options (/B /Q, etc.) | C-12 |
| MAKEFILE Analysis | C-13 |
| MAKEFILE Questions | C-14 |
| Compiler Optimizing | C-15 |
| Coding for Optimization | C-15 |
| Variable Type PCSZ | C-15 |
| Use of #define | C-15 |
| The #pragma Directive | C-16 |
| pragma chars, checkout, comment | C-17 |
| pragma langlvl | C-18 |
| pragma linkage | C-19 |
| pragma pack, seg16, stack16, subtitle, title . | C-20 |
| Dynamic Memory Allocation (malloc, etc) | C-21 |
| Buffering of stdout | C-22 |
| Compiler Macros including __MULTI__ | C-24 |
| Including .MSG files when Statically Linking | C-25 |
| Using DosCreateThread with Multithreaded Libraries | C-25 |
| Answers to Questions | C-26 |

IBM C Set Runtime Libraries

We'll discuss the six libraries which come with the IBM C Set Compiler for C programming. We will not discuss those for C++. Generally you do not have to deal with these by name, as compiler generates the names for you. But you do have to *select* the ones you want (or take the defaults) in your makefile.

| | <subsystem library> Rn | | | | * Ring 0 Code Possible! |
|-----|---|----------|----------------|---------------------------|-------------------------|
| [1] | DDE4NBS | LIB | Gd- Rn | Static Subsystem Library | |
| [2] | DDE4NBSI | LIB | Gd Rn | Dynamic Subsystem Library | |
| | DDE4NBS | DLL | | | |
| | DDE4NBSO | LIB | | | |
| | DDE4NBS | DEF | | | |
| | <Single Threaded Application Library> Gm- | | | | |
| [3] | DDE4SBS | LIB | Gd- Gm- | Static Singlethread | |
| | + | DDE4SBM | LIB Gd- Gm- Sm | Migration Library | |
| [4] | DDE4SBSI | LIB | Gd Gm- | Dynamic Singlethread | |
| | DDE4SBS | DLL | | | |
| | DDE4SBSO | LIB | | | |
| | DDE4SBS | DEF | | | |
| | + | DDE4SBMI | LIB Gd Gm- Sm | Migration Library | |
| | | DDE4SBM | DLL | | |
| | | DDE4SBM | DEF | | |
| | <Multi-Threaded Application Library> Gm | | | | |
| [5] | DDE4MBS | LIB | Gd- Gm | Static Multithread | |
| | + | DDE4MBM | LIB Gd- Gm Sm | Migration Library | |
| [6] | DDE4MBSI | LIB | Gd Gm | Dynamic Multithread | |
| | DDE4MBS | DLL | | | |
| | DDE4MBSO | LIB | | | |
| | DDE4MBS | DEF | | | |
| | + | DDE4MBMI | LIB Gd Gm Sm | Migration Library | |
| | | DDE4MBM | DLL | | |
| | | DDE4MBM | DEF | | |

You specify the subsystem library with **/Rn**. The default library is **/Gd- /Gm-** which is #3 above (single threaded statically linked application library) -- see page C-11. You can explicitly specify single-threaded or multi-threaded using **/Gm** or **/Gm-**; and static or dynamic with **/Gd** or **/Gd-**. You can also ask for the extra migration functions by specifying **/Sm**.

In the lab **makefile** we sometimes use the defaults which is perhaps bad practice since you cannot tell which libraries are used by looking at the makefile (unless you *know* the defaults).

Besides the C Runtime our applications dynamically link to the OS/2 2.1 system API. The *import library* for these is **OS2386.LIB**. If we are going to call some of the 16-bit API functions from our 32-bit program (making use of the *thunking* capability built into the IBM C Set Compiler), then we will also link to **OS2286.LIB**.

[1], [2] SubSystem Library

The subsystem libraries have no serialization, no initialization, and all routines are reentrant or documented not to be. Most are reentrant. The use of this library is (1) it is a subset of the full library, (2) it is small, it is fast to start up (no initialization), it comes in static or dynamic versions. If you are writing ANYTHING which only uses functions which are in this library, and you don't mind serializing yourself, we recommend using this library because it will be smaller in memory. Use **DosCreateThread** as `_beginthread()` is not supported. We used this library in Lab Project 2.

| LAB2D.EXE | file size | | |
|-----------|-----------|---------|--------------------|
| /Rn /Gd- | 33824 | static | subsystems library |
| /Rn /Gd+ | 24312 | dynamic | subsystems library |

The subsystems library is useful if you are developing code which is going to go into a device driver or a display driver or some sort -- partially running at ring 0. The startup code which the normal run-time has creates semaphores -- will not run at ring 0 or inside a device driver where these functions are not available.

In the subsystem library, you must use **DosCreateThread**. You then must provide your own serialization.

[3] Single Threaded Static

All the applications libraries have initialization code, are full function, and convert most operating system exceptions into easy to handle C signals. The single threaded statically linked application library may be used when you want independence from DLL modules which may or may not be in the target environment, and when you have only one thread in your application.

[4] Single Threaded Dynamic

If you have but one thread in your program you may link to the single threaded applications library. It can only be used with one thread per process but by any number of processes. The library uses instance initialization (a DLL topic we'll discuss on Thursday) to allocate a small block of memory for each process. In this block the library keeps state information (malloc tables, state tables, stuff that fopen uses) on a per process basis.

NOTE: If you are going to be using the multi-threaded dynamic for other processes, you might just as well use it for single threaded processes as well - it is quite fast and saves replication of DLL modules in RAM.

Thus, you may NOT want to use the single threaded dynamic.

[5] Multi-Threaded Static

Multi-Threaded Static: YOU MUST USE `_BEGINTHREAD`, and absolutely everything is serialized or reentrant. It uses a very fast serialization method so that the difference in performance between it and the single threaded library is for all intents unmeasurable except for the cases where there are collisions on serialization. You would use the static library instead of the dynamic library if you are not confident that the DLLs you need will be in the target environment.

[6] Multi-Threaded Dynamic

The Multi-Threaded Dynamic Library can handle many processes each of which have many threads - everything is taken care of. The performance is super unless there is a collision - the overhead to test for collision is less than a dozen cycles (if there is no collision). This library can be used for processes and for DLLs.

| TYPICAL.EXE file size | | | |
|-----------------------|------|-------|-----------------------------|
| /Gm | /Gd- | 65603 | static multithread library |
| /Gm | /Gd | 24768 | dynamic multithread library |

Migration Libraries

The migration libraries "front end" the other libraries. That is, if you specify `/Sm` in cases [3-6] you get an appropriate migration library as well as the specified library.

Typical MAKEFILE

The makefile below is from LAB2A. You can review makefile syntax by looking into the workshop Text Supplement under **NMAKE Examples** or typing **VIEW TOOLINFO NMAKE**. Note that we redirect the output to an error file (`main.err`, `tool.err`, etc). On the next few pages we will look at the compiler command line options.

```
browse.exe:
main.obj:  *.c
icc /c /Ss /Ti /Kb /Q /Rn /Fa *.c > *.err
type main.err

tool.obj:  *.c
icc /c /Ss /Ti /Kb /Q /Rn *.c > *.err
type tool.err

screen.obj: *.c
icc /c /Ss /Ti /Kb /Q /Rn *.c > *.err
type screen.err

browse.exe: main.obj tool.obj screen.obj main.def
link386 /DE /NOI /NOL /LI /BASE:0x10000 main tool screen,browse,,main;
```

Compiler Options - Output Files /F

Generally we do not use any of these /F options. In the previous example, we used /Fa (which is the same as /Fa+) to specify that we wanted to create main.asm. When that is done, you can look at the .ASM file to see the machine code emitted by the Compiler. Generally we need not do this (even as assembler programmers) as IPMD will show us a mixed listing and allow us to break on specific assembler instructions.

We do not use the /Fe option to specify that we want main.c converted to browse.exe. Instead, we use a spot in the option list of LINK386. See **VIEW TOOLINFO LINK386**.

| Output File Management Options | | |
|--------------------------------|--|---|
| OPTION | DESCRIPTION | DEFAULT |
| "/Fa[+ -]" "/Faname" | Produce and name an assembler listing file that has the source code as comments. | "/Fa-" Do not create an assembler listing file. |
| "/Fc[+ -]" | Perform syntax check only. | "/Fc-" |
| "/Fename" | Specify name of executable file or DLL. | Give the executable file the same name as the first source file, with the extension ".EXE" or ".DLL". |
| "/Flname" | Specify name of listing file. | "/Fl- " Give the listing the same file name as the source file, with the extension ".LST". |
| "/Fm[+ -]" | Produce a linker map file. | "/Fm-" |
| "/Fo[+ -]" | Create an object file. | "/Fo[+]" |

Compiler Options - Listing

Generally we never use the listing options, but I put them in here to show you that you can get some expanded listings if you so desire.

You can also specify titles using the `#pragma title` and `#pragma subtitle` directives, but these titles do not appear on the first page of the listing output. Try **VIEW DDE4LRM #pragma**.

| Listing Output Options | | |
|---------------------------|--|--|
| OPTION | DESCRIPTION | DEFAULT |
| <code>"/L[+ -]"</code> | Produce listing file. | <code>"/L-"</code> |
| <code>"/La[+ -]"</code> | Include a layout of all struct and union variables with offsets and lengths. | <code>"/La-"</code> Do not include a layout. |
| <code>"/Le[+ -]"</code> | Expand all macros. | <code>"/Le-"</code> Do not expand macros. |
| <code>"/Lf[+ -]"</code> | Set all listing options on or off. | <code>"/Lf-"</code> Set all listing options off. |
| <code>"/Li[+ -]"</code> | Expand user <code>#include</code> files. | <code>"/Li-"</code> Do not expand user <code>#include</code> files. |
| <code>"/Lj[+ -]"</code> | Expand user and system <code>#include</code> files. | <code>"/Lj-"</code> |
| <code>"/Lpnum"</code> | Set page length. | <code>"/Lp66"</code> |
| <code>"/Ls[+ -]"</code> | Include the source code. | <code>"/Ls-"</code> |
| <code>"/Lt"string"</code> | Set title string. | <code>"/Lt""</code> |
| <code>"/Lu"string"</code> | Set subtitle string. | <code>"/Lu""</code> |
| <code>"/Lx[+ -]"</code> | Generate a cross-reference table of variable, structure and function names that shows line numbers where names are declared. | <code>"/Lx-"</code> Do not generate a cross-reference table. |

Compiler Options -/K (lint like warnings)

The IBM C Set compilers have an excellent lint-like warning capability. We recommend that you always use /Kb when migrating to the IBM C Set compiler from another C compiler. We always use /Kb in the lab makefiles.

| Debugging Options - K Options | |
|-------------------------------|---|
| OPTION | DEFAULT |
| "/Ka[+ -]" | "/Ka-" Suppress messages about assignments that may cause a loss of precision. |
| "/Kb[+ -]" | "/Kb-" Suppress basic diagnostic messages. |
| "/Kc[+ -]" | "/Kc-" Suppress preprocessor warning messages. |
| "/Ke[+ -]" | "/Ke-" Suppress messages about "enum" usage. |
| "/Kf[+ -]" | "/Kf-" Set all diagnostic messages options off. |
| "/Kg[+ -]" | "/Kg-" Suppress messages about "goto" statements. |
| "/Ki[+ -]" | "/Ki-" Suppress messages about uninitialized variables. |
| "/Ko[+ -]" | "/Ko-" Suppress portability messages. |
| "/Kp[+ -]" | "/Kp-" Suppress messages about unused function parameters. |
| "/Kr[+ -]" | "/Kr-" Suppress messages about name mapping. |
| "/Kt[+ -]" | "/Kt-" Suppress preprocessor trace messages. |
| "/Kx[+ -]" | "/Kx-" Suppress messages about unreferenced external variables. |

Compiler Debugging Options

We always use the `/Ti` option in our labs to put debugging information into the `.OBJ` file, and then we use the `/DE` option to LINK386 to pass that on to the `.EXE` file. You might experiment with removing the `/DE` option to generate a smaller `.EXE`. In one case, where I was writing code in MASM I found that 50% of the size of my `.EXE` file was debugging information! Keeping the `/Ti` in your compiler options keeps the debug information available (making a bigger `.OBJ` file), and eliminating `/DE` from the LINK386 invocation drops the debugging information from the `.EXE` file.

The `W` (warning) errors are different than the `/K` (lint-like) error messages, so we keep them both in. Since the default for `/W` is `/W3`, we leave that parameter out of our makefiles.

| Debugging Options - /N /W /T | | |
|------------------------------|---|--|
| OPTION | DESCRIPTION | DEFAULT |
| <code>"/Nn"</code> | Set maximum number of errors before compilation aborts. | Set no limit on number of errors. |
| <code>"/Ti[+ -]"</code> | Generate C Set/2 debugger information. | <code>"/Ti-"</code> Do not generate debugger information. |
| <code>"/W[0 1 2 3]"</code> | Set the type of message the compiler produces and that causes the error count to increment. | <code>"/W3"</code> Produce all message types. |

`"/W0"` Produce only severe errors.
`"/W1"` Produce severe errors and errors.
`"/W2"` Produce severe errors, errors, and warnings.

Source Code Options

The only source code option we use is `/Ss`, which allows us to use the double slash for comments. For C programming, the use of a double slash is not recommended. We often do it as it seems to unclutter our source code and our transparencies. Use **VIEW DDE4LRM #pragma** to review `#pragma langlvl`, `#pragma pack`, `#pragma margins`, and `#pragma sequence`.

| Source Code Options | | |
|---|--|--|
| OPTION | DEFAULT | CHANGING DEFAULT |
| <code>/S[a e m 2]</code> | <code>"/Se"</code> Allow all language extensions except migration. | <code>"/Sa"</code> Conform to ANSI standards <code>"/Sm"</code> Allow migration extensions <code>"/S2"</code> Conform to SAA Level 2 |
| <code>/Sg[l][,<r *></code> <code>/Sg-</code> | <code>"/Sg-"</code> Do not set any margins: use the entire input file. | <code>"/Sg[l][,<r *]"</code> Set left margin to "l". The right margin can be the value r, or an asterisk can be used to denote no right margin. "l" and r must be between 1 and 65535 inclusive, and r must be greater than or equal to "l". |
| <code>/Sh[+ -]</code> | <code>"/Sh-"</code> Do not allow ddnames. | <code>"/Sh[+]"</code> Allow use of ddnames. |
| <code>/Sn[+ -]</code> | <code>"/Sn-"</code> Do not allow DBCS. | <code>"/Sn[+]"</code> Allow use of DBCS. |
| <code>/Sp[1 2 4]</code> | <code>"/Sp4"</code> Align structures and unions along 4-byte boundaries (normal alignment). | <code>"/Sp[1 2]"</code> Align structures and unions along 1-byte or 2-byte boundaries (structures and unions are considered to be "_Packed"). <code>"/Sp"</code> is equivalent to <code>"/Sp1"</code> . |
| <code>/Sq[l][,<r *></code> <code>/Sq-</code> | <code>"/Sq-"</code> Use no sequence numbers. | <code>"/Sq[l][,<r]"</code> Sequence numbers appear between columns "l" and r of each line in the input source code. |
| <code>/Ss[+ -]</code> | <code>"/Ss-"</code> Do not allow <code>//</code> for comments. | <code>"/Ss[+]"</code> Allow the double slash format to indicate comments. |

Code Generation Options /G /M /O /R

These are really the important ones. Note that the default is **/Re /Gm- /Gd- /O- /G3 /Gp-**. This means to use the standard application library, single threaded, statically linked, no optimization, 386 code (will run on 486). When we look at the execution trace analyzer \IBMCP\BIN\IXTRA.EXE we will see the use of **/Gp**.

| Code Generation Options | | |
|-------------------------|---|---|
| OPTION | DEFAULT | CHANGING DEFAULT |
| /Gd[+ -] | /Gd- Static Library | /Gd[+] Dynamic Library |
| /Ge[+ -] | /Ge[+] Build .EXE | /Ge- Build DLL |
| /Gh[+ -] | /Gh- no profiling hooks | /Gh[+] profiling hooks |
| /Gm[+ -] | /Gm- single thread lib | /Gm[+] multithread lib |
| /Gn[+ -] | /Gn- default libraries | /Gn[+] specify libraries |
| /Gp[+ -] | /Gp- Do not generate the code for protected DLLs. | /Gp[+] Generate the code to create a protected DLL. |
| /Gr[+ -] | /Gr- ring 3 | /Gr[+] ring 0 |
| /Gs[+ -] | /Gs- keep stack probes | /Gs[+] remove stack probes |
| /Gt[+ -] | /Gt- Do not map variables for 16 bit segments. | /Gt[+] Enable all variables to be passed to 16-bit functions. |
| /G3 /G4 | /G3 optimize for 386 | /G4 optimize for 486 |
| /Mp /Ms | /Mp optlink linkage | /Ms system linkage |
| /O[+ -] | /O- no optimization | /O[+] optimize code |
| /Re /Rn | /Re application library | /Rn subsystem library |

Other Compiler Options

The **/B** option allows you to pass a string to the linker. This is sometimes used to pass **/Li** to the linker to get line numbers in the map. Since we create our own makefiles and we run both the compiler and the linker out of our makefiles, generally we will never use the **/B** option.

| Other Options | | |
|--------------------------|---|---|
| OPTION | DEFAULT | CHANGING DEFAULT |
| <code>/B"options"</code> | <code>"/B""</code> Pass no parameters to the linker. | <code>"/B"options"</code> Pass options string to the linker as parameters. |
| <code>/C[+ -]</code> | <code>"/C-</code> Perform compile and link. | <code>"/C[+]"</code> Perform compile only, no link. |
| <code>/Hnum</code> | <code>"/H255"</code> Set the first 255 characters of external names to be significant. | <code>"/H"num</code> Set the first num characters of external names to be significant. The value of num must be between 6 and 255 inclusive. |
| <code>/J[+ -]</code> | <code>"/J[+]"</code> Set unspecified "char" variables to "unsigned char". | <code>"/J-</code> Set unspecified "char" variables to "signed char". |
| <code>/Q[+ -]</code> | <code>"/Q-</code> Display logo on "stderr". | <code>"/Q[+]"</code> Do not display logo. |
| <code>/V"string"</code> | <code>"/V""</code> Set no version string. | <code>"/V"string"</code> Set version string to string. The length of the string can be up to 256 characters. |

You can view these compiler options on line by typing **VIEW DDE4LRM Using Compiler Options**. You can then page forward from that point in the reference using the "F" key.

Makefile Analysis

We invoked link386 with the options and command line elements:

```
LINK386 [options] objfiles [,exefile, mapfile, libraries, deffile]
```

```
/DE /NOI /NOL /LI /BASE:0x10000 main tool screen,browse,,main;
```

You can review these by typing **VIEW TOOLINFO LINK386**. NOI says to preserve case sensitivity, NOL disables the sign-on banner, and /LI puts the line numbers into the map file. An .EXE file produced by the linker is relocatable, but OS/2 2.1 always starts the application's private memory space at 0x10000. By providing that information to LINK386 in the /BASE statement, certain *fixup* records are eliminated from the .EXE file making the loading faster and the .EXE file about 4% smaller.

```
browse.exe:
main.obj:  *.c
icc /c /Ss /Ti /Kb /Q /Rn /Fa *.c > *.err
type main.err

tool.obj:  *.c
icc /c /Ss /Ti /Kb /Q /Rn *.c > *.err
type tool.err

screen.obj: *.c
icc /c /Ss /Ti /Kb /Q /Rn *.c > *.err
type screen.err

browse.exe: main.obj tool.obj screen.obj main.def
link386 /DE /NOI /NOL /LI /BASE:0x10000 main tool screen,browse,,main;
```

The /Fa option to the compiler caused MAIN.ASM to be produced, part of which is shown below. Notice the **INCLUDELIB** statements which tell the linker exactly which of the C runtime libraries to include.

```

TITLE    MAIN.C
.386
.387
INCLUDELIB os2386.lib
INCLUDELIB dde4nbs.lib
CODE32   SEGMENT DWORD USE32 PUBLIC 'CODE'
```

Makefile Questions

1] Why do we explicitly specify **/Kb** but not specify **/W3** when we invoke the compiler? What is the purpose of the **/Q** option? What is **\$***?

a]

2] Why do we route the output of the compiler to **\$*.err**? What is the result of this if there are severe errors in the compile?

a]

3] What is the purpose of **/BASE:0x10000**. What other steps can be taken to enhance the **.EXE** file in this manner?

a]

4] When using **NMAKE**, you can define a macro (string substitution) by writing **macroname = new string**. You can reference the macro anytime with the syntax **\$(macroname)**. Define a macro called **cOptions** and show how you would use it in the makefile on page C-13. Why do this?

a]

[] Check your answers on page C-26.

IBM C Set Compiler Optimizing

The IBM C Set Compiler optimizes quite well -- it is one of the best features of the compiler. It produces state of the art code. In general, do not optimize when running IPMD (the debugger). If you do optimize, you can still break on function entry and function exit, but that's about it. The transformations that the optimizer performs are of such an extensive nature that the relationship between source code and object code is very very blurry after optimization. We unroll loops, get rid of calls when they can be eliminated (jumps, etc.).

General hints to stay out of the way of the optimizer: use local variables instead of static or global. If the variable is only used in the procedure not only is it faster but the optimizer knows who can and who cannot change it. Don't take the address of a local variable because the optimizer then has to make more pessimistic assumptions about it.

If you have a function which receives a pointer, and the function does not change what this pointer points at, put a CONST in the prototype function because the optimizer will then know that you don't change it.

```
void foo (const char *)
```

const is a promise to the compiler that you will not assign it in foo.

If you have loop indices make them local variables, and do not take the address of a loop index -- then it is hard to put it in a register.

Variable Type PCSZ

This material is from PROGREF21: If a function takes as a parameter a string that is not changed by the function, the string parameter can be declared as a "const" string, or a PCSZ. PCSZ is defined in the C++ header files as a "const" pointer to a NULL-delimited string. The "const" means that the function will not change the contents of the string.

Declaring the parameter as PCSZ informs the C++ compiler that the function will not change the string. Therefore, the compiler simply passes a pointer to the string in the function parameter list. If the parameter is declared as a normal PSZ (not "const"), the compiler assumes that the function might change the string. Under these circumstances the compiler will add code to make a copy of the string then pass a pointer to the copy, rather than passing a pointer to the original string.

A smaller, faster executable is often produced if the data item passed in a parameter list is declared as "const". See **VIEW PROGREF21 PCSZ data type**.

Use of #define

When declaring the name of shared memory object, such as \sharemem\ourlab\tuesday, use a #define rather than a PSZ. That way you can put the item in an .H file shared by many source files, and the compiler knows that this is a constant string.

The #pragma Directive

A pragma is an implementation defined instruction to the compiler. It has the general form given below, where character sequence is a series of characters giving a specific compiler instruction and arguments, if any.

```
>>—#—pragma—character_sequence—><
```



The character sequence on a pragma is not subject to macro substitutions. More than one pragma construct can be specified on a single #pragma directive.

The following pragmas are available:

- alloc_text
- chars
- checkout
- comment
- data_seg
- handler
- langlvl
- linkage
- map
- margins
- pack
- page
- pagesize
- seg16
- sequence
- skip
- stack16
- strings
- subtitle
- title

Pragma **handler** is used to register the C runtime exception handler `_exception`. It can also be used in conjunction with **map** to register your own handler. Pragmas **linkage**, **seg16** and **stack16** are used with compiler thinking. The linkage pragma may be replaced with in line statements, such as:

```
void _Far16 _Cdecl wait_key (void (* _Far16 _Cdecl)(char, char));
```

Appendix T on Thinking and appendix E on Exception handling will deal with some of these issues.

Compiler Pragmas

>> #pragma chars (signed) <<
 unsigned

>> #pragma checkout (resume) <<
 suspend

>> #pragma comment (compiler
 date
 timestamp
 copyright
 user , -"characters") <<

#pragma chars allows you to override the default, which is unsigned.

#pragma checkout allows you to suspend the diagnostics performed by the /K options during specific portions of your program, and then resume the same level of diagnostics at some later point in the file. See **VIEW DDE4LRM checkout**. Note therein that the /K options should not be used for new code!

In the case of #pragma comment:

compiler - the name and version of the compiler is emitted into the end of the generated object module.

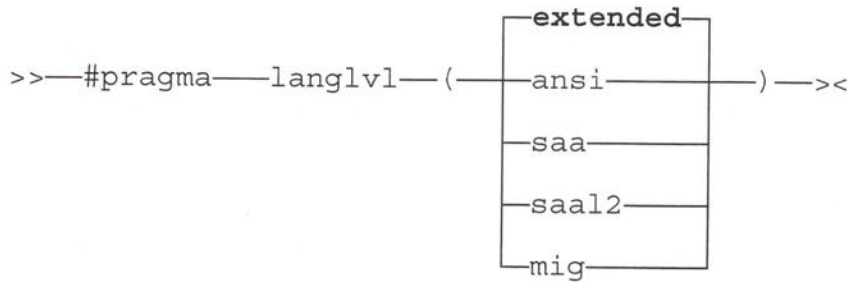
date - the date and time of compilation is emitted into the end of the generated object module.

timestamp - the last modification date and time of the source is emitted into the end of the generated object module.

copyright - the text specified by the character field is placed by the compiler into the generated object module and is loaded into memory when the program is run.

user - the text specified by the character field is placed by the compiler into the generated object but is not loaded into memory when the program is run.

#pragma langlvl



This `#pragma` directive can be specified only once in your source file, and must appear before any C code. The language level can also be set using the `/Sa`, `/S2`, `/Se`, and `/Sm`

The compiler defines preprocessor variables that are used in header files to define the language level. The options are as follows:

`ansi`: Defines the preprocessor variables `__ANSI__` and `__STDC__`. Allows only language constructs that conform to ANSI C standards.

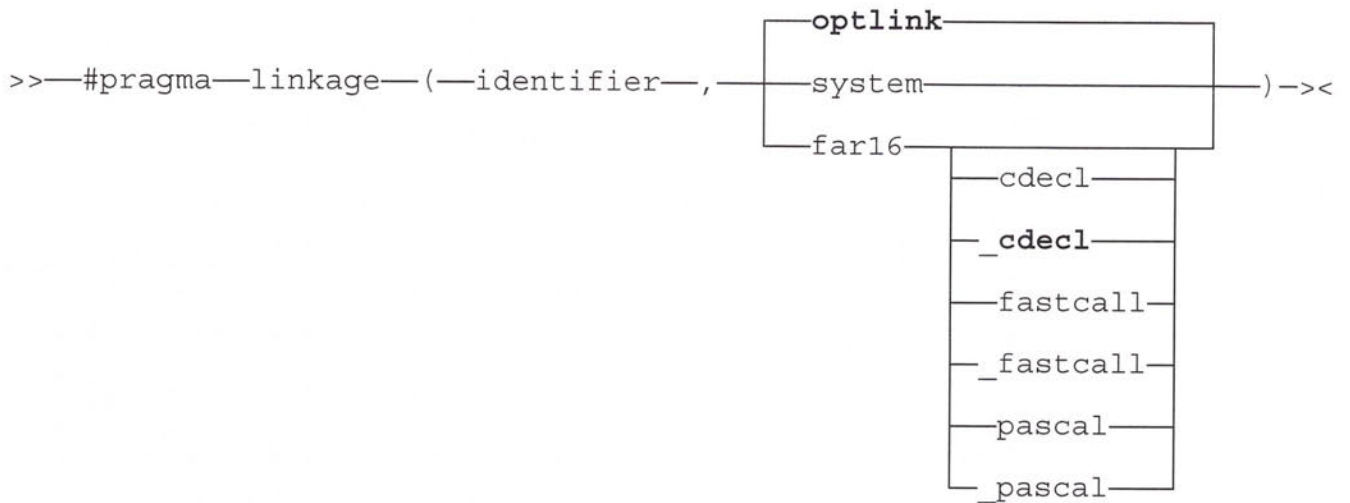
`saa`: Defines the preprocessor variables `__SAA__` and `__SAA_L2__`. Allows only language constructs that conform to the most recent level of SAA C standards (currently Level 2). These include ANSI C constructs.

`saal2`: Defines the preprocessor variable `__SAA_L2__`. Allows only language constructs that conform to SAA Level 2 C standards. These include ANSI C constructs.

`extended`: Defines the preprocessor variable `__EXTENDED__`. Allows ANSI and SAA C constructs and C Set/2 Standard extensions. Migration extensions are not allowed.

`mig`: Defines the preprocessor variable `__MIG__`. Allows ANSI and SAA C constructs, and all C Set/2 extensions, both Standard and Migration.

#pragma linkage



Linkage has to do with the way parameters are passed to procedures. All OS/2 system functions (all the Dos API we use, etc) are declared APIENTRY in the function prototypes in the header files. APIENTRY is defined to be **_System** in OS2DEF.H.

The linkage pragma (or equivalent linkage keywords) is the key element in Thinking (at least from our point of view, as applications programmers).

The C Set/2 default linkage is `optlink`, which is a convention specific to the C Set/2 product. However, if your program calls OS/2 APIs, you must use the `system` linkage convention, which is standard for all OS/2 applications. Remember that threads created with `DosCreateThread` need **system** linkage, whereas threads created with `_beginthread` need **optlink** (the default).

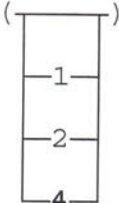
The **far16** linkage conventions indicates that a function has a 16-bit linkage type. The `cdecl` and `_cdecl` options are equivalent. The underscore is optional, and is accepted for compatibility with C/2 `cdecl` linkage declarations. Similarly, `pascal` and `_pascal` are equivalent and specify C/2 pascal linkage, and `fastcall` and `_fastcall` specify Microsoft `_fastcall` linkage. If `far16` is specified without a parameter, `_cdecl` linkage is used.

You can also use linkage keywords to specify the linkage type for a function. Linkage keywords are easier to use than the `#pragma linkage` directive, and let you declare both the function and its linkage type in one statement. See the example on page C-16.

You can use compiler options to explicitly set the linkage type to `optlink` (`/Mp`) or to change the default to `system` linkage (`/Ms`). These options are described in Code Generation Options. However, if a linkage keyword or `#pragma linkage` directive is specified, it overrides the compiler option.

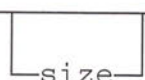
#pragmas continued

```
>> #pragma pack ( ) <<
```

A diagram showing a vertical rectangle representing a stack of memory. The rectangle is divided into three horizontal sections. The top section is labeled '1', the middle section is labeled '2', and the bottom section is labeled '4'. This illustrates how the #pragma pack directive affects the alignment of variables in a structure.

```
>> #pragma seg16 ( identifier ) <<
```

```
>> #pragma stack16 ( ) <<
```

A diagram showing a horizontal line representing a stack size. A bracket below the line is labeled 'size', indicating the length of the stack.

```
>> #pragma subtitle ( " subtitle " ) <<
```

```
>> #pragma title ( " title " ) <<
```

The `#pragma pack` directive specifies the alignment rules to use for the structures and unions that follow it. Packing on 4-byte boundaries (the default) is tuned to the 32-bit wide bus trading space for performance. Try **VIEW DDE4LRM PACK**.

`#pragma seg16` allows you to declare a pointer as a 16:16 pointer. That is, the compiler will maintain the contents of that pointer as a 16:16 pointer. If `#pragma seg16` is used on variables of a structure type, the pointers inside that structure are not automatically qualified as usable by 16-bit programs. If you want the pointers in the structure qualified as such, you must declare them using the `_Seg16` type qualifier. We will discuss this Appendix T.

`#pragma stack16` allows you to specify how much stack a 16-bit function that you call will have. We'll look at the mechanics of that in Appendix T as well.

Dynamic Memory Allocation

```
void *malloc(size_t size);    <- size in byte requested
```

malloc reserves a block of storage of **size** bytes. malloc returns a pointer to the space, or NULL if the space is not available. The storage is private to the process. The memory is committed when it is allocated.

```
void *free (void *ptr);      <- address of block to free
```

return storage allocated with malloc to malloc pool

```
int _heapmin(void);         -> 0 == ok, -1 == fail
```

The `_heapmin` function returns all unused memory from the run-time heap to the operating system. Not SAA.

```
void *alloca(size_t size)    <- bytes requested
```

This function may be used inside a procedure to allocate memory from the stack. It is very fast, but the memory is deallocated when you end the procedure. The memory may not be shared with other processes. Not SAA.

malloc uses **DosAllocMem** under the covers, so that the memory is private to the process. malloc asks for a minimum of 64K and keeps a pool of free memory from which it allocates buffers. **free** returns memory to the malloc pool, but malloc does not do a **DosFreeMem** (to return the memory to OS/2) unless (1) you do a **heapmin** and (2) there is a free 64K memory object that malloc can release back to OS/2.

alloca is another C function to allocate memory. There is no corresponding free since the memory is allocated out of the stack of the calling function. **alloca** is extremely fast because it generally just means moving the stack pointer. However, **alloca** is allocating from a finite resource (the thread's stack) whereas **malloc** has access to the full process address space by using **DosAllocMem** directly. Allocations made with malloc last the life of the process (or until you do a free), but allocations made with alloca last only the life of the function in which the allocation is made.

Lab project three part one in the advanced class labs (see Appendix L) does some performance measurement of malloc, alloca, DosAllocMem and DosSubAllocMem. That lab uses the 32 millisecond system clock to time-stamp presentation manager messages as the measurement tool. Better tools include IXTRA (the IBM Execution Trace Analyzer which comes with C Set++) and ASDT - the application and system debug tool from IBM Lexington which is released on the Developer's Connection.

Buffering of stdout

- * IBM C Set/2 buffers stdout by default
- * IBM C/2 and Microsoft C 5.X do not

```
#include <stdio.h>
int main (void)
{
    printf ("To continue, press the <Enter> key:");
    getchar ();
}
```

What's wrong with this program?

When is buffering of stdout useful?

Buffering of stdout means that the output directed to stdout is not immediately written to stdout but instead is written to a buffer and copied to stdout when either the buffer fills or when a newline character is received.

In the example above, you do not see the message asking for the enter key until you hit the enter key. When the enter (newline) character is echoed, the message comes out. This is disconcerting and makes examples in K&R not work!

If standard out is redirected, the buffering of standard out can be useful. But when standard out is the display, you (the programmer) must be aware of the effects.

Buffering of stdout - Three "solutions"

```
#include <stdio.h>
int main (void)
{
    printf ("To continue, press the <Enter> key: \n");
    getchar ();
}
```

```
#include <stdio.h>
int main (void)
{
    setbuf (stdout, NULL);
    printf ("To continue, press the <Enter> key: ");
    getchar ();
}
```

```
#include <stdio.h>
int main (void)
{
    printf ("To continue, press the <Enter> key: ");
    fflush (stdout);
    getchar ();
}
```

There are three ways to work around the problem on the previous page -- the problem that the message "Hit ENTER" does not print out until you actually do hit enter. First, you can include a newline (`\n`) character in your message.

Second, you can specify that you will provide the buffer for standard out and that there is not one! This is done using `setbuf (stdout, NULL)`. This statement must be done before the first character transfer to standard out.

If you want to leave buffering to standard out enabled, and you want to see a line which does not have a newline character at the end, then you can execute `fflush(stdout)` which causes an immediate dump of the stdout buffer to the stdout device.

Some Macros Defined by the Compiler

There are a number of macros defined by the compiler. A partial list of them appears below. We use `__MULTI__` in the labs. To see a full list, type `VIEW DDE4LRM ADDITIONAL`.

`__cplusplus`

Set to the integer 1. Indicates the product is a C++ compiler. This macro is valid for C++ programs only.

`__DLL__`

Indicates code for a DLL is being compiled. Defined using the `/Ge-` compiler option.

`__MULTI__`

Indicates multithread code is being generated. Defined using the `/Gm` compiler option.

`__OS2__`

Set to the integer 1. Indicates the product is an OS/2 compiler.

`__SPC__`

Indicates the subsystem libraries are being used. Defined using the `/Rn` compiler option.

Static Linking and .MSG Files

At certain times the compiler runtime will print messages to STDOUT. These include the error messages from the C exception handler. If you have the .MSG files on disk with the normal pathing (to \IBMCP\HELP) then statically and dynamically linked EXEs and DLLs will find the messages. If you produce a statically linked EXE and run it on a system which does not have these .MSG files, you will get a series of "message not found" messages.

Generally this will not occur when you are dynamically linking, because you have the C "stuff" on your disk when you execute. But if you statically link and ship off your EXEs to another (user) environment, you may see this problem.

The message file used at runtime by the C/C++ runtime is DDE4.MSG. To stay within the license, you MUST bind it to your app; you can't ship it....

One problem, multiple solutions:

1. If you have WorkFrame/2 Version 2.1 installed, the following command will bind the runtime messages to your executable:

```
DDE3MSGB xxx.exe d:\path\HELP\DDE4.MSG
```

Where:

xxx.exe is your executable
d:\path is where you installed the C++ compiler

2. If you have WorkFrame/2 Version 1.1, create a project that contains DDE4.MSG, any object file, and which creates your target. Go into makefile generation, and select the object file and DDE4.MSG, and the actions LINK and MESSAGE BIND. Makefile generation will then generate a make file which contains the command and input file for the MSGBIND utility from the toolkit.
3. If you do not have workframe, you can look up the runtime message numbers, and the instructions for using MSGBIND, and build all the files yourself. The runtime message numbers are documented in the "IBM C/C++ Tools Online Language Reference" (in the C Set++ folder), and MSGBIND is documented in the "Tools Reference" (in the Toolkit Information folder).

Using DosCreateThread with Multithreaded Libraries

To use DosCreateThread with the multi-threaded libraries:

1. Use #pragma handler() on the new thread
2. call _fpreset() to set up the NPX
3. Make the function _System linkage
4. End the thread with _endthread() to clean up library storage.

**May not be portable across compiler revisions*

Answers to Questions

<page C-13>

1] Why do we explicitly specify **/Kb** but not specify **/W3** when we invoke the compiler? What is the purpose of the **/Q** option? What is **\$***?

a] The default is no lint testing, so we use **/Kb** to get that extra validation of our source code. The default is **/W3**, which is the highest warning level. The **/Q** option suppresses the banner. The **\$*** is a *macro* which means "take the name from the left part of the item to be made by this statement. If we are providing the dependency list for main.obj, then **\$*** would be "main".

2] Why do we route the output of the compiler to **\$*.err**? What is the result of this if there are severe errors in the compile?

a] Since we are not using workframe (although you are free to do so), we save the error messages to a file so that we can bring up the errors and the source file with the editor. For example, you could say **EPM main.c main.err** and put both files in the ring. A file **M.CMD** in \OS20LABS says **E m*.c m*.err**. There is also a **L.CMD** which says **E lab*.c lab*.err**. It just a trick. Note that if you are using workframe, you *do not* want to route the output to a file but want it to go directly to the screen.

If there are severe compiler errors, **NMAKE** will terminate with a 12 error and you will not see the errors until you bring up the **.err** files. Had we not done this redirection, you would see the errors on the screen.

3] What is the purpose of **/BASE:0x10000**. What other steps can be taken to enhance the **.EXE** file in this manner?

a] The **/BASE:0x10000** causes the linker to remove relocation fix-up records. We do this because we know that OS/2 2.1 places the **.EXE** in at **0x10000**. You can actually see the layout of your **.EXE** by typing **EXEHDR BROWSE.EXE**. The end result is that the **.EXE** file is smaller and loads faster.

You can also enhance your **.EXE** file by removing debugging information, specifying **/EXEPACK** to make the file smaller, and (of course) by dynamically linking to the C runtime. Further, if you allow the compiler to optimize, there is a significant shrinkage of the code size and a speed-up in execution.

Generally during debugging we do not want to optimize and we do want debugging information in the **.EXE** file. At some point we want to change all this.

Answers to Questions

- 4] When using NMAKE, you can define a macro (string substitution) by writing **macroname = new string**. You can reference the macro anytime with the syntax `$(macroname)`. Define a macro called `cOptions` and show how you would use it in the makefile on page C-13. Why do this?

```
cOptions = /c /Ss /Ti /Kb /Q /Rn  
browse.exe:  
main.obj:  $.c  
    icc $(cOptions) /Fa $.c > $.err  
    type main.err  
tool.obj:  $.c  
    icc $(cOptions) $.c > $.err  
    type tool.err  
screen.obj: $.c  
    icc $(cOptions) $.c > $.err  
    type screen.err  
browse.exe: main.obj tool.obj screen.obj main.def  
    link386 /DE /NOI /NOL /LI /BASE:0x10000 main tool screen,browse,,main;
```

- a] You might do this so that you could easily replace the compiler options when you switch from debugging mode to production mode.

OS/2 2.1 for Software Developers

Appendix D

The OS/2 2.1 Desktop

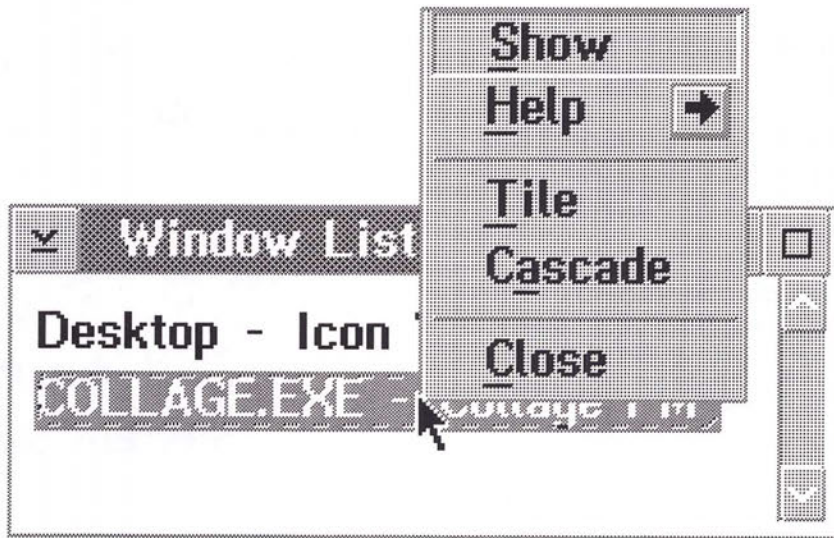


by Charles R. Chernack

| | |
|--|------|
| Chapter D - The OS/2 2.1 Desktop | D-1 |
| The Window List | D-3 |
| The Desktop Selection Menu | D-4 |
| Desktop - Settings | D-5 |
| Icons Flowed Invisible | D-5 |
| Sort Order and Background Color | D-6 |
| Setting up the Desktop - Command Prompts, Drives | D-7 |
| Starting Sessions with the Start Command | D-8 |
| STARTUP.CMD and the Startup Folder | D-9 |
| Start New Window Each Selection | D-10 |
| Cleaning up the Command Prompts Folder | D-11 |
| Removing little-used Objects | D-11 |
| Adding a Workshop Help Object | D-11 |
| Work Areas | D-13 |

The Window List

This section contains some notes on using the OS/2 2.1 desktop. They have been derived from experience with the desktop, rather than from book learning.



CTRL+ESC will bring up the Window List. You can also bring up the window list by clicking **both** mouse buttons at the same time while the mouse pointer is on the desktop (background).

You can change the size and position of the window list. OS/2 will remember this information. You can override the memorized position by bringing up the window list with the mouse (the window list is centered around the mouse position). To move the window list put the mouse into the "Window List" title bar, and drag the window while holding down the left mouse button. To size the window list, put the mouse on an edge or corner of the window list and size while holding down the left mouse button.

The mouse buttons are technically mouse button 1 and mouse button 2. You can set the mouse for right or left handed operation using **System Setup, Mouse, and Setup**. We will use left and right to indicate mouse button 1 and mouse button 2.

You can select an item in the window list by keying in the first letter of the item, using the up and down arrows, or by clicking once using the left mouse button (LMB). You can show an item in the window list by hitting ENTER, double clicking with the LMB, or bringing up the pop-up and selecting **Show**.

The pop-up will come up if you click right in the window list while an item is selected, or if you use SHIFT+F10. The pop-up is most useful to easily close windows directly from the window list. You can dismiss the pop-up with ESC.

In subsequent pages, we will see how to name our OS/2 and DOS window and full screen sessions, so that their icons will be easily recognized, and so that they will be easy to select from the window list. We will also see how to have OS/2 place the icons on the desktop instead of in the minimized window viewer.

The Desktop Context Menu

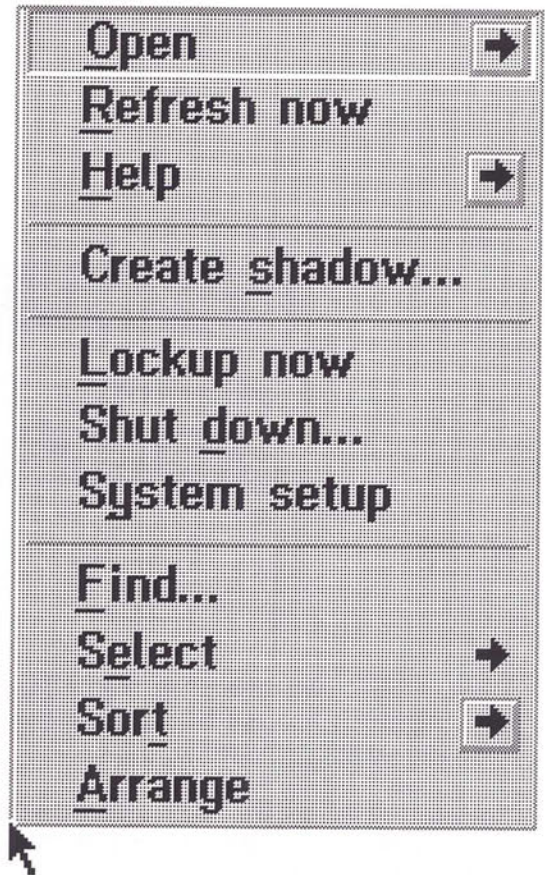
If you click the RMB on the desktop, you will get a desktop menu. The primary use of this menu is to **Open** the desktop settings, lock the desktop, **Shut down** the system, and quickly get to **System setup**.

The sequence "click RMB" on the desktop and "type d" seems to be the quickest way to perform a system **Shut down**. It is thus useful!

Alternatively you can do a keyboard shut down. This requires that you select the desktop from the window list, hit the spacebar to deselect all desktop objects, and then press SHIFT + F10 to bring up the desktop context menu.

If you have not previously set up to lock the desktop, the first time you select **Lockup now** you will be asked for a password (to unlock the desktop). This is a convenient way to leave your system without shutting it down, and to prevent others from using your OS/2 system. You unlock the desktop by typing your password in. This can be different than your keyboard (power-up) password.

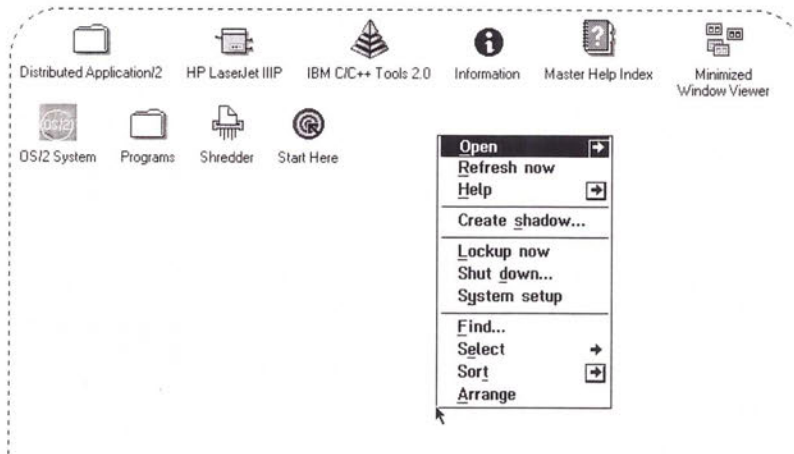
You can also enter **System setup** quickly from the desktop menu, rather than hunting for it directly in the OS/2 System folder.



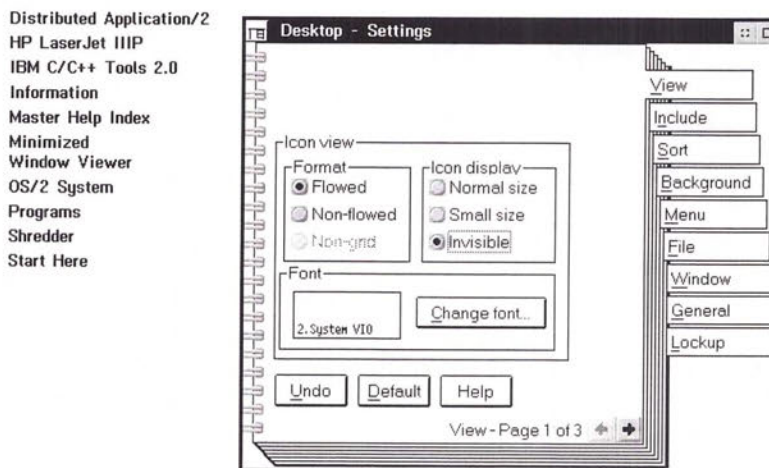
Desktop - Settings

You can bring up the **Desktop Settings** by clicking right on the desktop and then clicking on the arrow to the right of **Open** and then clicking on **Settings**. In general, you can do anything with the keyboard that you can do with the mouse, so you ought to try it both ways. Use the cursor keys (arrows) to go to **Open** and then right-arrow to **Settings**. You can also go to **Open** by pressing the O key.

In the picture below, the icons are shown as **Non-grid Normal size**. If you select **Non-flowed** you get a single vertical column of icons, whereas **flowed** provides multiple columns. The bottom figure shows how I usually arrange my desktop.

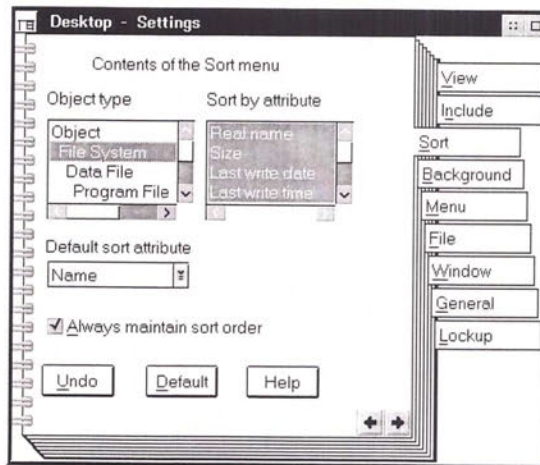


I like to arrange my desktop as a vertical list of folder names, rather than as a set of visible folders. OS/2 2.1 allows you to do this by setting the **Icon View** to **Flowed Invisible**. If you do this, your desktop will look like the one below. Now, while we have **Desktop - Settings** up, let's change the **Background** color and specify a **Sort** by name order.

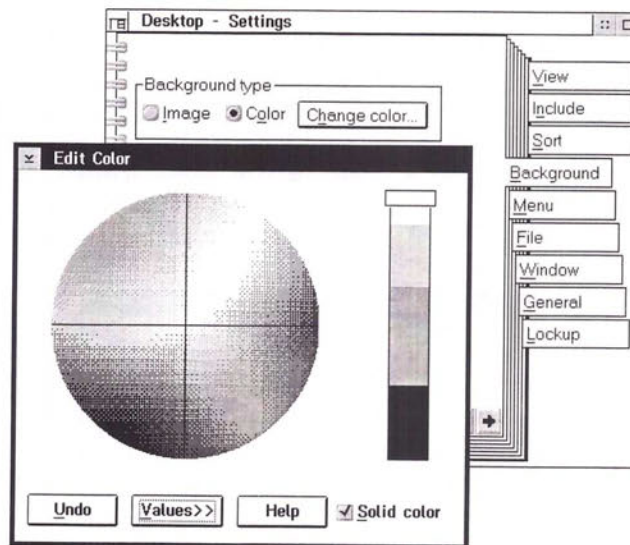


Desktop - Settings (Continued)

I like to keep the folder list on my desktop in alphabetic order. You can do this by selecting **Sort** and clicking on **Always maintain sort order**.



You can set the background color of the desktop by selecting **Background** and then **Change color**. You can also select a bitmap image for the background, but I find those distracting. I prefer a color to a bitmap.

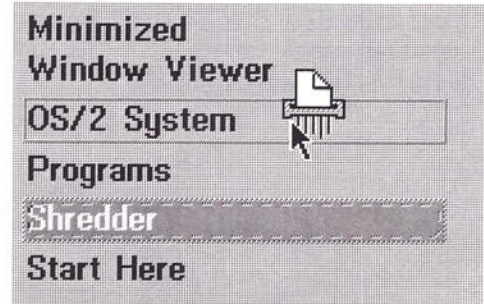


I often select a **Solid color**. The color or pattern you are selecting shows up in the little rectangle which moves up and down the bar like a slider. When you have chosen your background color, select it by closing the Edit Color dialog (double click in the upper left hand corner).

Drag Folders into OS/2 System

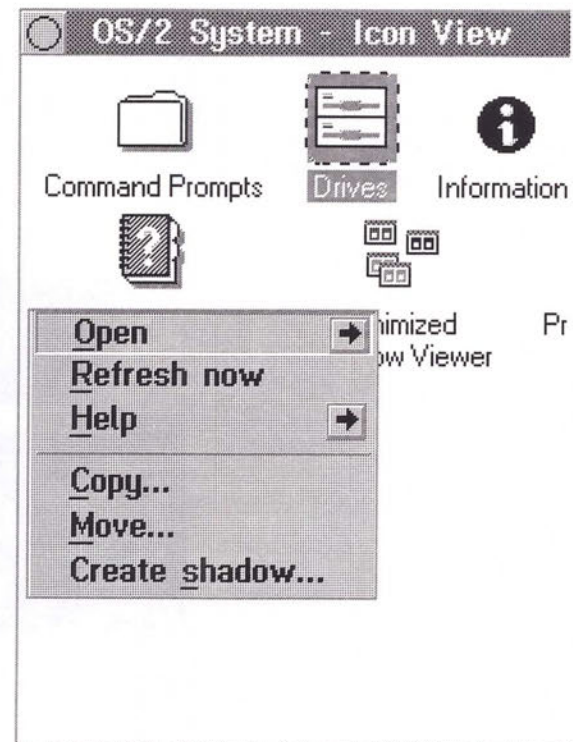
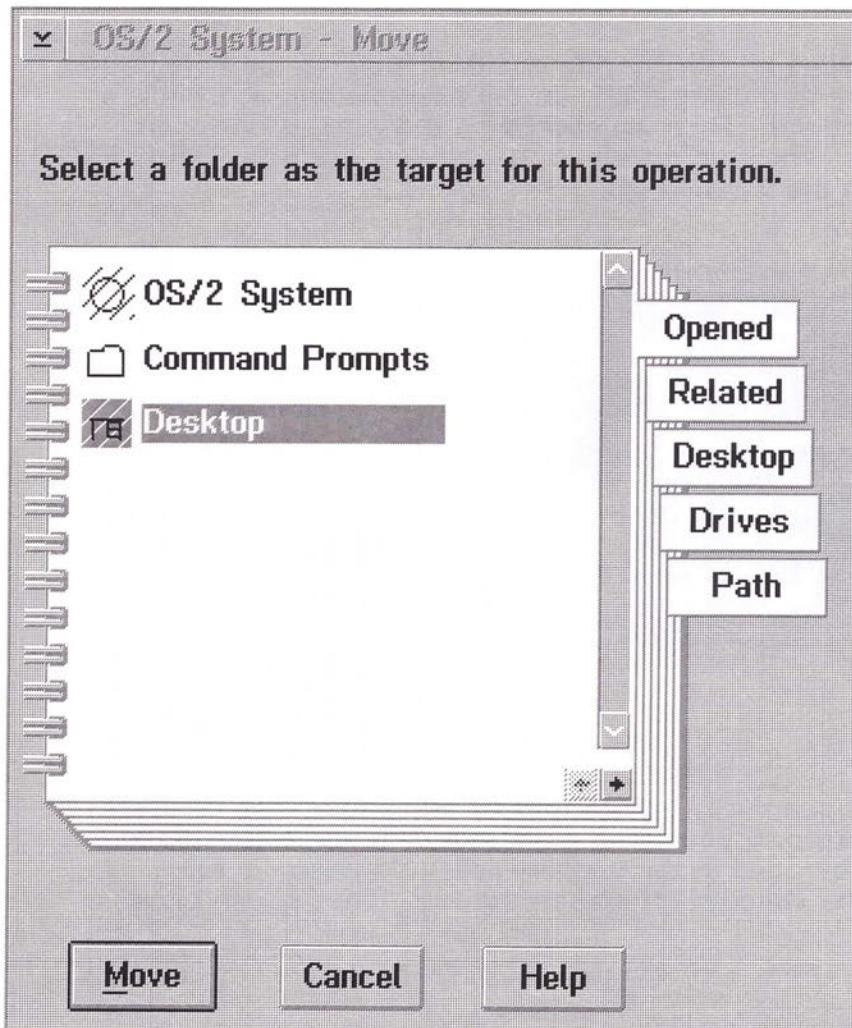
I generally remove folders from the desktop by dragging them into OS/2 System. You can click right on the folder and delete it, but if it is a folder that I might use someday, I generally drag and drop it into OS/2 System. Select the folder with the left mouse button, and then drag it with the right mouse button into the destination folder.

In this example, I am dragging the Shredder and dropping it into the OS/2 System folder.



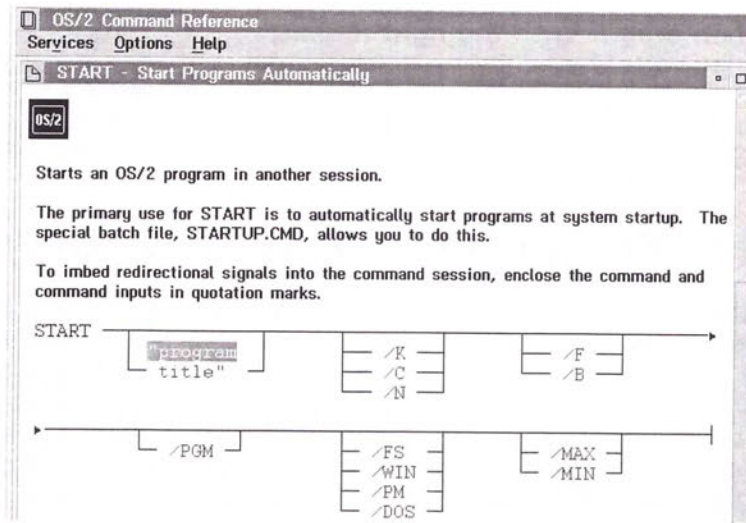
Copy Command Prompts to Desktop, Move Drives to Desktop

The drives folder is the file manager. I always **Move** that out of OS/2 System and put it on the desktop. I also **Copy** command prompts to the desktop. That's because I am going to change command prompts and would like to keep a copy of the original in OS/2 System. To copy, move, or create a shadow of an object in a folder, you can click right on the object which brings up the open/copy/move menu. You can also move an object by dragging it with the RMB. The target folders using the menu (as below) are all open folders. However, when dragging an object with the RMB you can drop it into any folder.

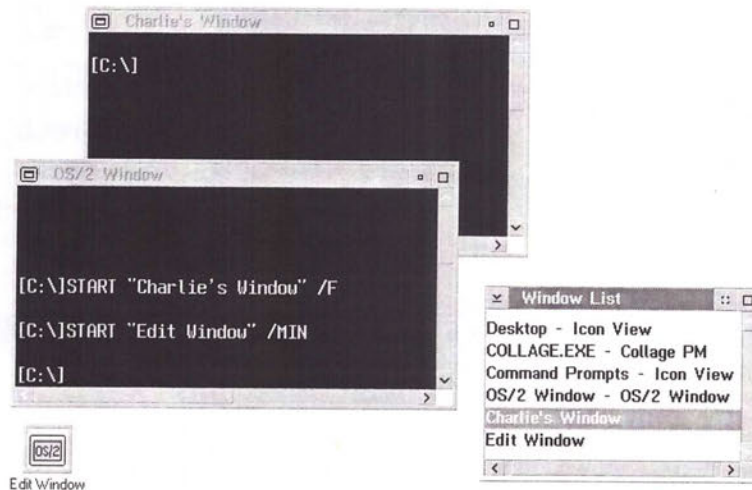


Starting Sessions with the Start Command

You can use the START command from an OS/2 window or an OS/2 full screen to start another session. You can view the syntax of the START command by typing HELP START (shown below).



Start "Charlie's Window" /F will start an OS/2 window with a "title" in the Foreground. You can also start a window minimized using /MIN. In the example below, Charlie's Window was started in the foreground. Then we clicked on the initial window and started "Edit Window" minimized. Then we pulled up the window list with CTRL + ESC. Note that the window titles are in the window list.



When a window has a name, you can easily switch to it by bringing up the Window List even if you have a very busy desktop. If you name a window, the name appears in the title bar, on the icon for the window, and in the window list. If you name a full screen you still get the name in the window list and on the icon. Thus naming is a way to rationalize the desktop.

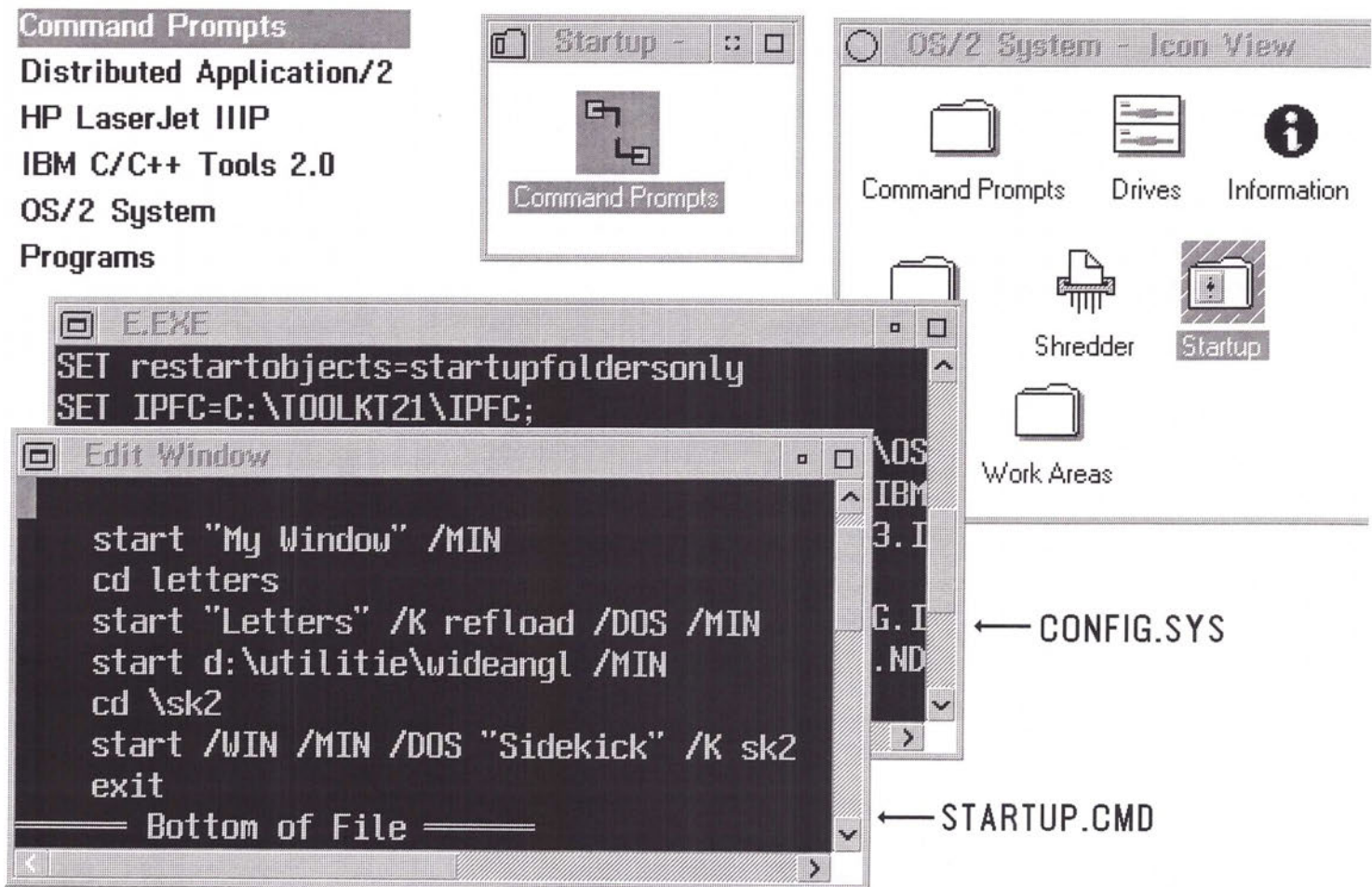
STARTUP.CMD and the Startup Folder

On power-up OS/2 2.1 opens any folders and starts any program objects which you have placed into the Startup Folder. In addition, OS/2 2.1 executes file \STARTUP.CMD, which may include START commands.

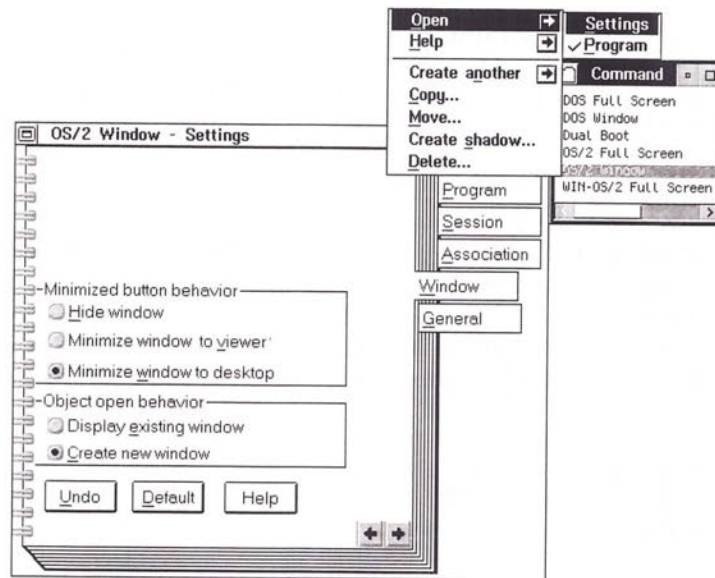
Thus there are two other ways to start sessions. You can open the startup folder (from OS/2 System). Put shadows of folders to be opened on power-up into the startup folder. I use shadows in the startup-folder so that if I change the folders the changed folder (rather than an old copy) will come up on power-up.

My own STARTUP.CMD contains a statement to bring up a named OS/2 window minimized, and to start a minimized DOS session and initialize the session by running batch file reflowd.bat, and to start Sidekick in a minimized DOS Window. An EXIT at the end of STARTUP.CMD eliminates the startup OS/2 session once the STARTUP.CMD is processed.

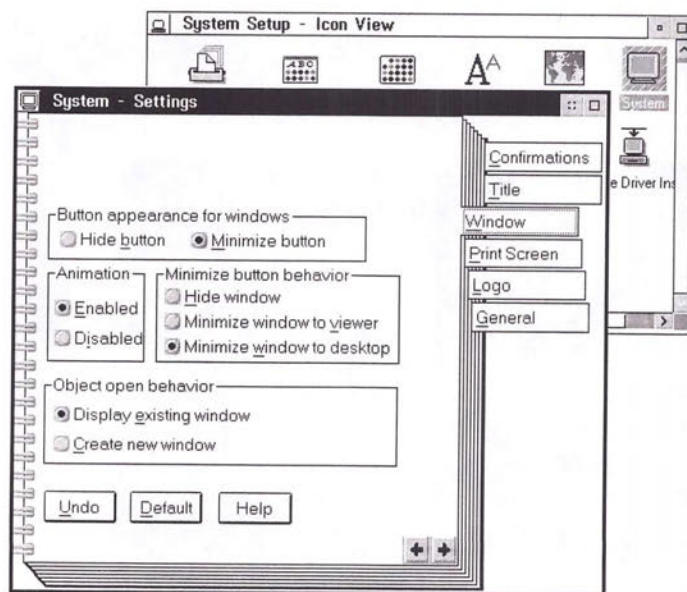
In the figure below, you can see a shadow of Command Prompts in the Startup folder which comes from OS/2 System. I also modify CONFIG.SYS to include the statement **SET restartobjects = startupfolderonly**. This prevents objects which were up on the desktop from automatically starting when I power-up OS/2. I prefer that as I always start with the same uncluttered desktop. Later we will look at Work Areas, a technique to bring up a set of folders or programs associated with a specific project.



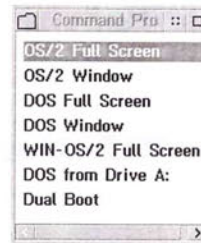
Once we place a copy of the **Command Prompts** folder onto the desktop, it is time to modify the default settings. If you start an OS/2 Window and then try to start another from the Command Prompts folder, you will simply activate the OS/2 window which you currently have open. This is fine for Compuserve (a program you only want to run one copy of), but not fine for OS/2 windows. Click right on **OS/2 Window** in the **Command Prompts** folder, and open the **Settings**. Select **Window** and **Minimize Window to desktop** and **Create new window**. The result will be that every time you click on OS/2 Window you will get a new one, and the windows will minimize to the desktop.



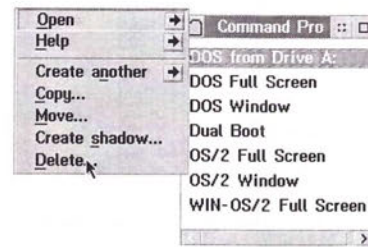
If you click right on the desktop and bring up the desktop menu, you can then select **System Setup** and **System**. If you double click on System you will get the **System - Settings** dialog which will allow you to set Logo timeout, etc. In the **Window** selection you can change the global defaults for all windows. I usually leave this alone and then for individual items I will go in and set "Create new window (as above).



Once we have copied the command prompts folder to the desktop we then set the icons to flowed invisible. I generally set the background to yellow, and I set sort to **Always maintain sort order** (that has been done in the second figure below). I size and position the command prompts folder, and then close it so that OS/2 will remember the size and position.

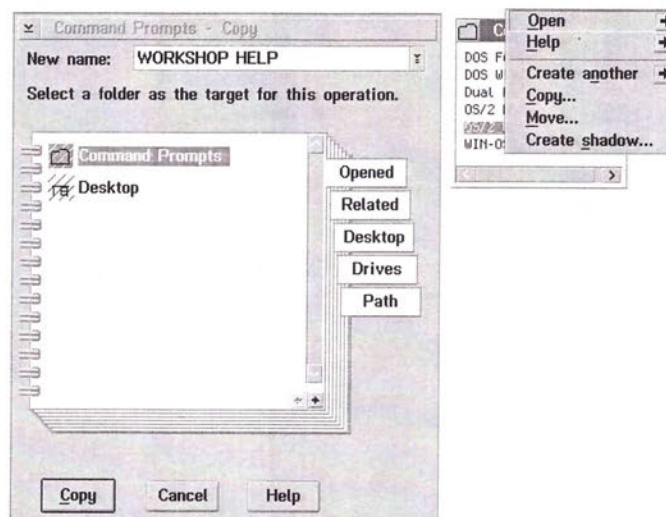


I then click right on **Dos from Drive A:** and delete it. Remember that if I want that feature I can always get it from the original **Command Prompts** folder still in **OS/2 System**.



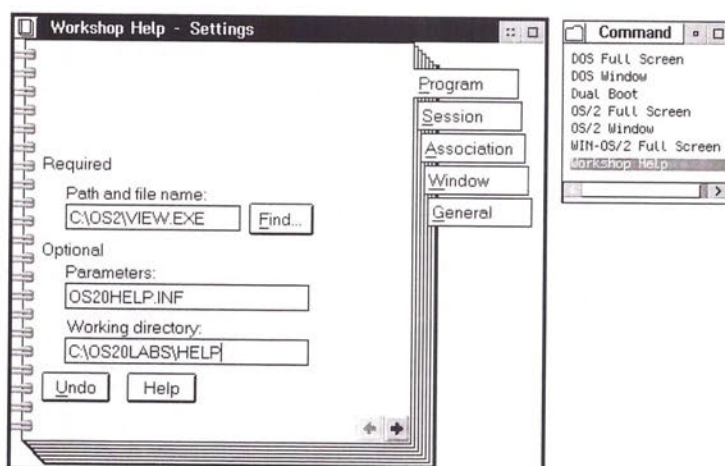
Adding Workshop Help

I am now going to make an entry in **Command Prompts** called **Workshop Help**. This entry will start the workshop help. To create the entry I click right on **OS/2 Window** and **Copy** it to the same folder **Command Prompts** changing its title to **Workshop Help**.

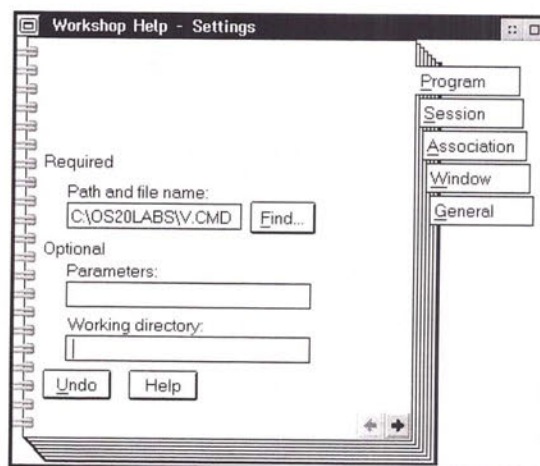


Adding Workshop Help (continued)

I open the **Settings** of **Workshop Help** by clicking right on it. We will demonstrate two different ways to fill in these settings. In the first method, set the **Path and file name** to C:\OS2\VIEW.EXE, the **Parameters** to OS2OHELP.INF, and the **Working directory** to C:\OS2OLABS\HELP. VIEW.EXE is the program which brings up and displays all of the on line references. Normally it uses the BOOKSHELF environment variable to find the path to the .INF files. However, the workshop help in C:\OS2OLABS\HELP is not in the BOOKSHELF environment, so we set the **Working Directory**.



Besides the Workshop Help, we want to view the Text Supplement and we want to start the Review Questions. We can do all of these operations by running command file V.CMD which is in C:\OS2OLABS. It is a powerful technique to put a .CMD file name into the **Path and file Name**.



Work Areas

A **work area** is a very useful capability of the OS/2 desktop. You place a set of program objects which you use for a specific task or project and place them into a work area folder.

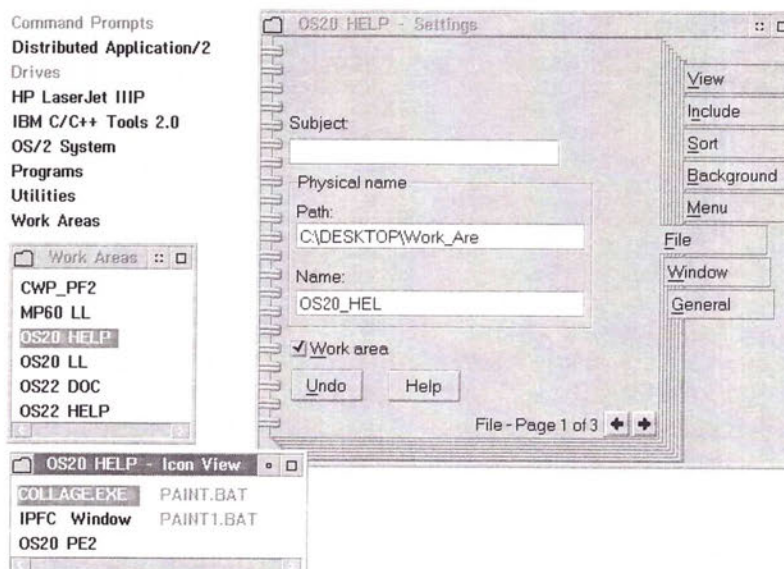
Generally when you open objects from within a folder, you can then close the folder and the objects remain open. That is, there is no *parent-child* relationship between folders and the objects created in them.

Any folder can be made into a **Work area**. This is done by going to the **File** selection in the folder's settings, and checking **Work area**. The operation of this "work area" folder is identical to other folders, with an exception: if you minimize the work area then all objects opened out of the folder will disappear off the screen (effectively minimizing into the folder). When you restore the work area folder then all previously open objects will be restored to their previous size and position.

If you close the work area folder then all objects opened out of that folder will be closed. When you reopen the work area folder than all objects previously open will be reopened.

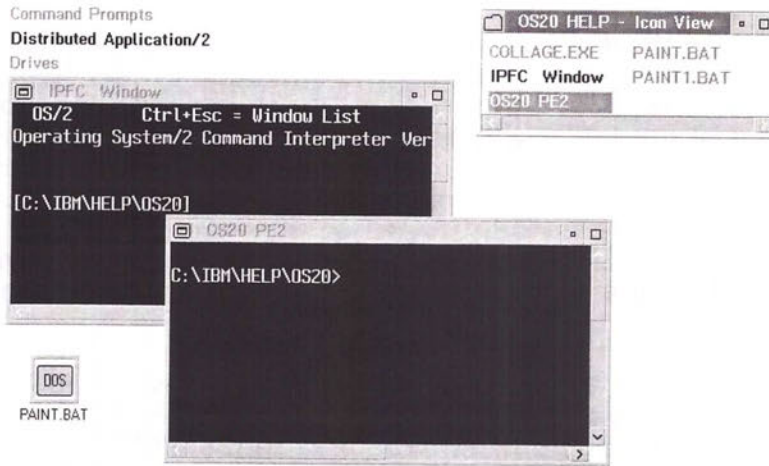
Now what is the use of this? Well, when using my computer I have various tasks I do. Mostly they involve preparing class material. I have a work area for OS20 HELP which contains a Dos Edit window in the appropriate directory, an OS/2 IPF Compile window in the same directory, a screen snapshot program and a paint program. That is, the set of objects I use to work on that project all appear in one work area folder. Thus it is very easy to restore my working environment for a particular project.

I have one folder titled Work Areas which is NOT a work area. It holds other folders which are work areas. For example, OS20 HELP contains 5 objects which I use when preparing the workshop help. OS20 LL contains the Linkway Live tools I use for preparing the review questions. Note that OS20 HELP is a work area.



Work Areas (continued)

The first figure below shows three objects open out of the OS20 HELP work area. PAINT.BAT is a full screen DOS session containing Publisher's Paintbrush. In the second figure below, the work area OS20 HELP was minimized taking with it all of the objects that had been opened from that folder.



OS/2 2.1 for Software Developers

Appendix DLL

Dynamic Link Libraries

Some material in this chapter
courtesy IBM Toronto Lab

by Charles R. Chernack

| | |
|---|--------|
| Chapter DLL - Dynamic Linking | DLL-1 |
| Technical Background | DLL-3 |
| Variable Storage in the Application and the DLL | DLL-9 |
| Static, Load Time and Run Time Binding | DLL-12 |
| Review Questions | DLL-13 |
| Toolkit On Line Reference - the DATA statement | DLL-15 |
| DLL Instance Data - DATA MULTIPLE NONSHARED | DLL-16 |
| DLL Initialization and Termination | DLL-17 |
| _CRT_Init Case 1 - Static Linking no DLLs | DLL-18 |
| _CRT_Init Case 2 - Dynamic Linking no DLLs | DLL-19 |
| _CRT_Init Case 3 - Dynamic Linking, user DLLs | DLL-20 |
| DLL Initializaiton and Termination | DLL-21 |
| _CRT_Init Case 4 - Exporting the C Runtime | DLL-22 |
| _CRT_Init and Statically Linked DLL | DLL-24 |
| LIBRARY [] INITGLOBAL TERMGLOBAL | DLL-25 |
| DLL Instance Data - DATA SINGLE SHARED | DLL-26 |
| Using a DLL to Manage a Resource | DLL-27 |
| Answers to Questions | DLL-28 |

Technical Background

The dynamic link library capability is an essential component of the operating system. Subroutines or API required by running programs need not be linked into the EXE file of the application, but may be stored as DLL modules which can be attached by each application at (1) load time; or (2) at run time.

DLL routines are not programs. They are procedures which run in the context of the calling thread and calling process. Any number of processes can attach a DLL. The use of DLLs provides for code sharing between processes, resulting in smaller .EXE files on disk and lower run-time RAM loading.

OS/2 Code Sharing

If you load multiple copies of the same .EXE file from the same directory, OS/2 automatically shares the code and read-only data. But if two different applications are loaded, then of course OS/2 does not share the code. OS/2 has no way of knowing that statically linked library routines inside those applications are, in fact, common procedures.

If you have common procedures which you would like to share among applications, rather than statically linking those into each application, you may make DLL modules out of those routines. The code is then automatically shared between all attaching processes.

Multi-Threading and DLLs

The DLL is written in the same way as any procedure you might statically link into your process. If you want that procedure to be called by multiple threads from the same process, you need to make it reentrant (all variables in the stack).

DLL Data Segments

Generally each time a DLL is attached, any static or global data declared in the DLL is put into unique *instance data* for that DLL for that process. Thus the fact that the DLL is being used by many processes is irrelevant. DATA MULTIPLE NONSHARED gives you unique copies of the DLL instance data, on a per-process basis.

DLL Coupling with the Application

The DLL can export code labels and pointers to global variables. Thus an application can directly call procedures within the DLL, and it can directly access variables within the DLL. (Variable referencing is very very seldom used). However, the application cannot export symbols to the DLL -- they must be passed in via calls to procedures within the DLL.

How a DLL is Made

A DLL is a creature of the OS/2 linker (LINK386) and the OS/2 Loader. These packages work together to create the final runnable program in memory. If the linker is not going to finish the job -- not going to statically link procedures into the .EXE file -- then it must pass information to the loader to do the job at load time.

To make a DLL you should:

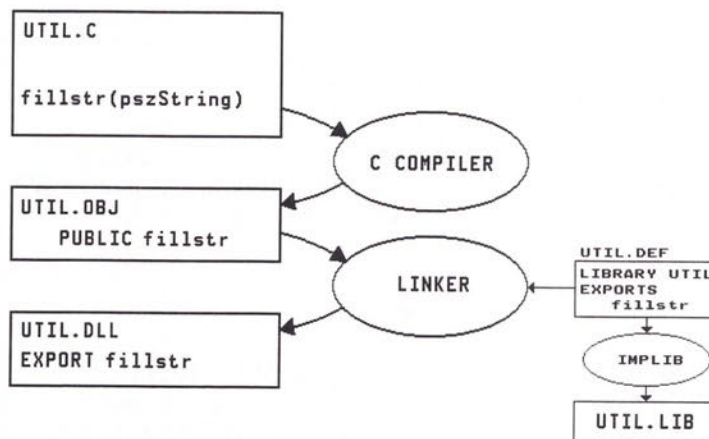
Tell the compiler: specify /Ge- when you compile the DLL.

Tell the linker to make a .DLL rather than a .EXE: put LIBRARY rather than NAME in the .DEF file

Provide the linker and loader with information they need: list the names of all procedure entry points you want to make available to the application using an EXPORTS statement in the DLL's module definition file.

Optionally make an import library from the module definition file.

In the example below, source file UTIL.C is converted to UTIL.DLL and the the module definition file UTIL.DEF is converted to UTIL.LIB, the import library:



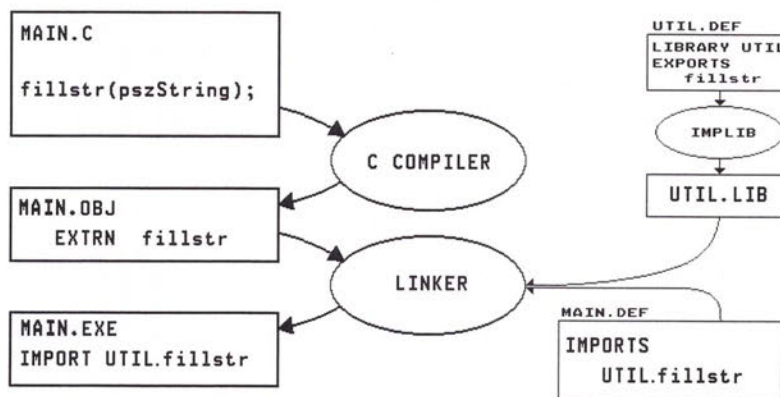
The Linker can produce an EXE or a DLL file. The LIBRARY statement in the module definition file causes the linker to produce a .DLL file. The EXPORTS statement in the module definition file lists which symbols are to be made available to routines which call the DLL.

At load time the OS/2 loader will search the LIBPATH for the requested DLLs.

Making the .EXE

In order to make an .EXE file which uses the DLL, you must satisfy LINK386 so that the procedures in the DLL are not considered undefined externals at link time. You can do this by adding IMPORT statements in the module definition file used to create the EXE, or by using IMPLIB on the DLL's module definition file to create an import library.

You also must put the DLL in the LIBPATH so that the OS/2 loader can find it. We set LIBPATH to .;C:\OS2\DLL;... The "." allows us to put the DLLs we create in our own directory.



Procedure fillstr will not be available at LINK time as it will come from a DLL when the application is loaded. In order to satisfy the linker (and to not get an unresolved external message) the linker needs to be told that procedure fillstr will be acquired later.

The module definition file for the application may specify that fillstr will be imported from a DLL using the IMPORT statement. The IMPORT statement provides the module name and procedure name: in this example it specifies that module UTIL.DLL will be providing procedure fillstr.

Alternatively, an IMPORT LIBRARY may be prepared which contains the same information: that procedure fillstr will be provided in module UTIL.DLL. The import library is prepared using \TOOLKT21\OS2BIN\IMPLIB.EXE. The input to IMPLIB is the module definition file which was used to create UTIL.DLL. See VIEW TOOLINFO IMPLIB.

DLL Initialization

Generally there is no *need* for DLL Initialization. If you want initialization, you can specify that a routine you provide called `_DLL_InitTerm` is to be called by the operating system twice: once for initialization and once for termination.

(1) If you have a `_DLL_InitTerm` routine it will be called by default every time a new process attaches or detaches (so long as you have specified *instance* initialization).

(2) If you do not have such a routine, the C Set Compiler will provide a default one.

(3) You can specify that rather than call the `_DLL_InitTerm` function every time any process attaches or detaches, you only want it called once the first time any process attaches (global initialization) and that you want it called only once when the last process detaches (global termination). This is the default and generally you do *not* want this option.

DLL Termination

If your DLL has allocated resources on behalf of the application, such as opening files or driving a printer or carrying on a dialog with the user, the DLL may wish to know that the application has terminated. You can use the standard **DosExitList** and **atexit** handlers.

While `DosExitList` handlers work (a carryover from 1.3), a DLL which has registered an `exitlist` will not be freed when `DosFreeModule` is used. It will hang around until the process terminates.

`_DLL_InitTerm` will also be called upon termination (either of any attaching process or when *all* processes using the DLL terminate).

C Runtime Initialization

If the DLL is prepared using the `/Gd` option, then the DLL itself will dynamically link to the C runtime. The DLL version of the C runtime has its own `_DLL_InitTerm` function and initializes itself. However, this initialization may occur after the `_DLL_InitTerm` function in your DLL is called. Thus, if you want to use C Applications Library functions inside the `init` case of `_DLL_InitTerm`, then you must call `_CRT_Init` yourself.

DLL_InitTerm, LIBRARY dllname INITINSTANCE TERMINSTANCE

A DLL or an EXE file has a record which specifies the initial value of CS:EIP. You can see this if you run EXEHDR on the DLL or EXE. In the case of a DLL, the fact that the CS:EIP field is non-zero means two things: (1) it supplies the address for the `_DLL_InitTerm` function, and (2) it acts as a flag to say that there is an initialization - termination entry point.

Thus `_DLL_InitTerm` is NOT an entry "name" known to OS/2; it is known to the compiler and the compiler puts a record in the .OBJ file specifying the address of `_DLL_InitTerm` as the CS:EIP entry value.

There are two other flags in the DLL file which can be viewed using EXEHDR. These specify whether there is instance or global initialization, and whether there is instance or global termination.

| | |
|--------------|--|
| Library: | LAB4CDLL |
| Module type: | Dynamic link library Per-process initialization Global termination |

#pragma data_seg (segmentname), SEGMENTS statement in .DEF file

DATA MULTIPLE NONSHARED in the module definition file for the DLL specifies that the DGROUP data (global data and static variables) will be replicated for each process which attaches the DLL. But let's say that you want to have some data which is shared between all copies of the DLL, and some data which is private to each instance of the DLL. You may do that by creating new named data "segments" using `#pragma data_seg`, and then you may specify MULTIPLE NONSHARED or SINGLE SHARED in the SEGMENTS statement of the module definition file.

The figures below are from the advanced class lab4c. The C code on the left from LAB4CDLL.C shows the use of `#pragma data_seg` to draw lines in the global data declarations of a program to force data items into specific named segments. The .DEF file on the right shows the use of the SEGMENTS statement to create "common" or shared instance data as well as to create process specific copies of a data item. Bottom line you control your DATA segment using the DATA statement in your module definition file, and you can create other segments with desired characteristics as shown below.

```
#pragma data_seg(globdata)
ULONG ulGlobal_Shared;

#pragma data_seg(instdata)
ULONG ulGlobal_Instance;

#pragma data_seg()
```

from LAB4CDLL.C

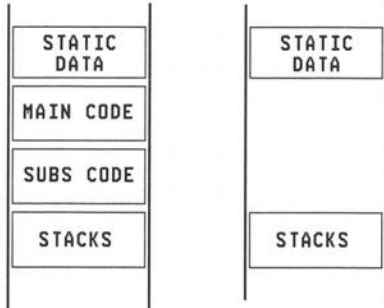
```
LIBRARY LAB4CDLL INITINSTANCE

SEGMENTS
    globdata CLASS 'DATA' SHARED
    instdata CLASS 'DATA' NONSHARED
```

from LAB4CDLL.DEF

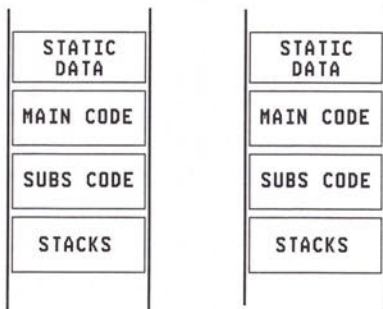
Review of Code/Data Sharing

OS/2 CODE SHARING
CASE 1: RUN SAME .EXE TWICE



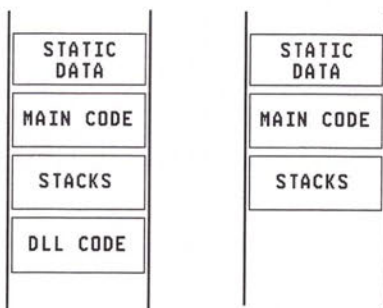
If the same .EXE file is loaded twice, the code will only be loaded once into RAM. Data and Stacks associated with the .EXE file will be replicated for each instance of the .EXE file, but the code (being a pure segment) may be shared between all instances of the same application.

OS/2 CODE SHARING
CASE 2: DIFFERENT EXEs WITH COMMON SUBS



If two different .EXE files are loaded, and they happen to have some common subroutines statically linked into the .EXE, there is no way for OS/2 to tell that the code is in fact common. Thus two copies of the code are loaded.

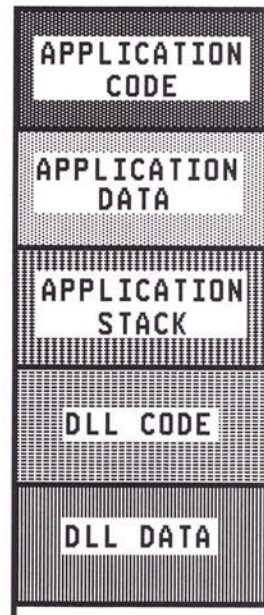
OS/2 CODE SHARING
CASE 3: DIFFERENT EXEs WITH DLL SUBS



If two different .EXE files are loaded, and they attach a DLL which has common subroutines, then OS/2 will only load one copy of the DLL into RAM.

Variable Storage in Application and DLL

```
CHAR a;  
main (void)  
{  
  CHAR b;  
  static CHAR c;  
  dllsub(a,b,c);  
}
```



```
CHAR dlldata[20];  
dllsub (CHAR a, CHAR b  
{  
  CHAR dlla;  
  a = 'x';  
}
```

Analyze where variables `a`, `b` and `c` from the application are stored. Can the DLL reference these variables by name?

Analyze where array `dlldata` and variable `dlla` are stored. Can the application reference these variables by name?

ANSWERS:

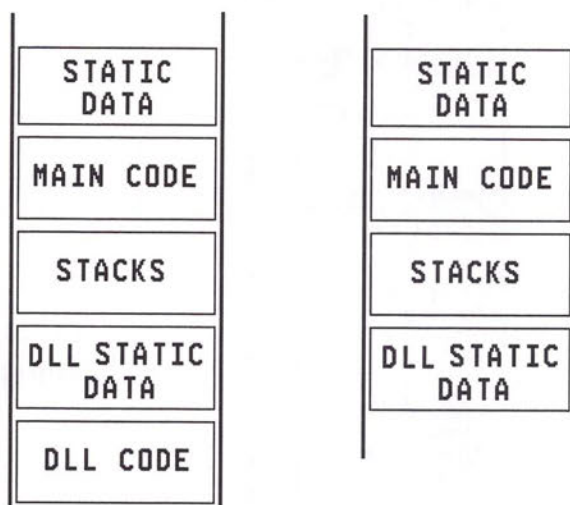
Variables `a` and `c` are stored in the application data. Variable `b` is stored in the application stack. The DLL cannot reference symbols in the application by name. There are two reasons for this: (1) you cannot export symbols from the application; and (2) even if you could, it would be impossible to reference it from the DLL. For example, let's say that variable "a" could be exported. Some applications would have a variable "a" and some would not. If applications had a variable "a", it would likely be in a different linear address for each application. There would be no way that the *fixup* could be handled. That is, the address of variable `a` could not be resolved when the DLL was loaded it.

Array `dlldata` is in DLL Data (DLL Static Data), and variable `dlla` is in the stack of the application thread which called the DLL. The application could reference `dlldata` "by name" only if the DLL exported a pointer to that static variable.

DLL Shared Code, Instance Data

OS/2 CODE SHARING

CASE 4: SEPARATE EXEs, DLLs with STATIC DATA

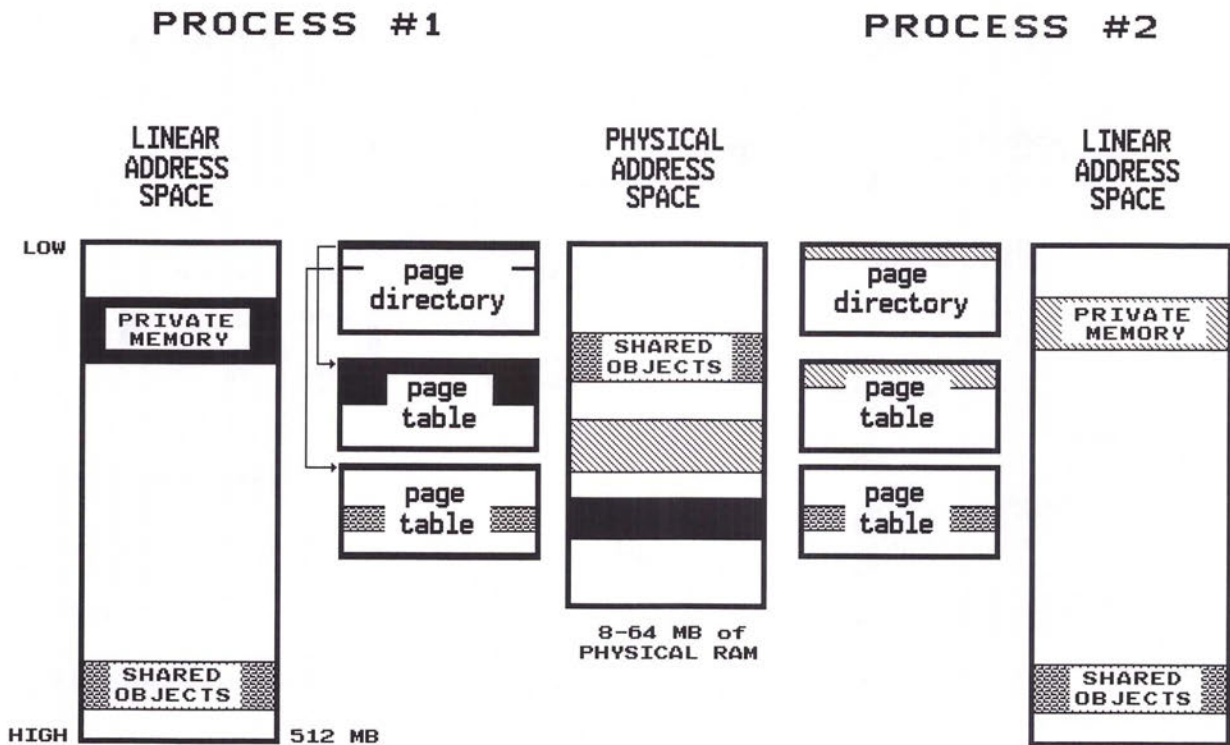


If two different .EXE files are loaded, and they attach a DLL which has common subroutines, then OS/2 will only load one copy of the DLL into RAM. If the DLL itself has any global variables or any static data, then there will be DLL INSTANCE DATA (DLL static data).

Since there are unique copies of the instance data on a per-process basis, the DLL may be called from different processes with data isolation. However, if the DLL is called by multiple threads in one process, then classic methods of protecting single-copy data (semaphores, critical section, etc.) must be used.

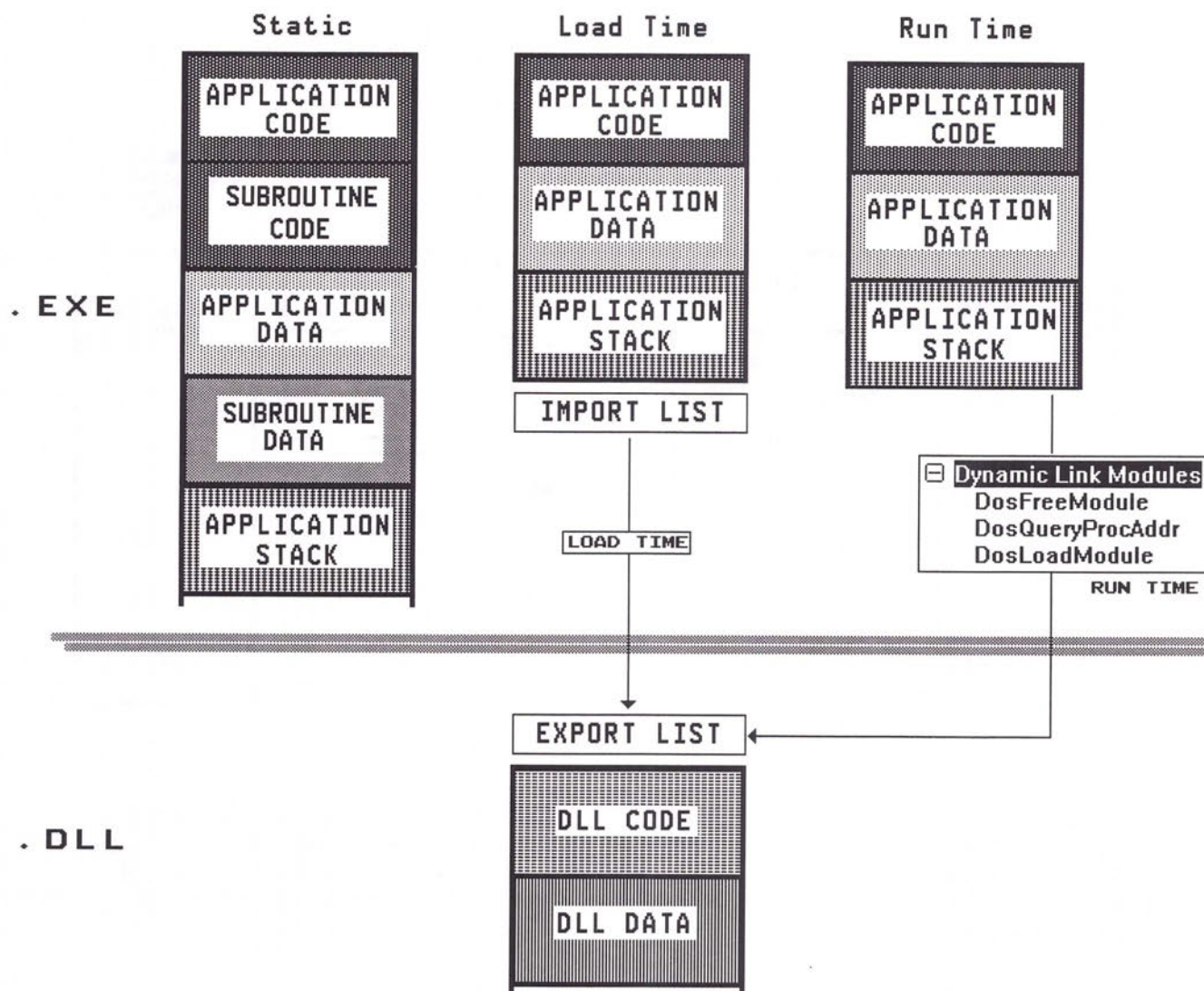
Thus the subroutines in the DLL operate as if there were only one process attaching the DLL, even if many processes attach the DLL. However, like any subroutine, the programmer must be careful of reentrancy issues. The reentrancy issue is always with you in a multi-threaded process.

Shared Code and Data in OS/2 2.1



Memory objects allocated using `DosAllocSharedMem`, DLL code segments, and DLL data segments are stored in the high address space. The same rules apply to both: once an object is assigned a linear address, that address space is reserved for all processes. The address space is only released when the reference count of the memory object goes to zero.

Static, Load Time, and Run Time Binding



With static linking, the linker binds the subroutines and their data into the .EXE file. The resulting .EXE file is self contained.

With load time binding, the .EXE file lacks required subroutines and those entry points are listed in the .EXE file as import records. The OS/2 loader follows the LIBPATH to find the corresponding DLL modules.

With run time binding, the .EXE file does not specify the imports. At run time the application can attach the required DLLs using `DosLoadModule`; find the addresses of the required subroutines using `DosQueryProcAddr`, and use indirect calls through pointers to actually call the subroutines. When the DLL is no longer needed, it can be detached by using `DosFreeModule`.

The DLL will *not* detach after a `DosFreeModule` if the DLL has used `DosExitList` to register an exitlist handler. This is one reason why `_DLL_InitTerm` is preferred.

Questions on Code and Data Sharing

1] When an application executes **DosAllocSharedMem** linear address space is used in which processes? When will that linear address space be freed?

a]

2] When an application attaches a DLL, linear address space is used in which processes? When will that linear address space be freed?

a]

3] Under what circumstances will code be shared between two processes which use static linking? What do you have to do to get this to happen?

a]

4] Under OS/2 1.3, how could you share statically bound data segments between processes? Why would you want to do this? What is an easier (and perhaps safer) way to share data?

a]

5] How can you get .EXE files with different names and *some* common code to share the code?

a]

Questions on Code and Data Sharing

6] What is the primary benefit of writing "library" or common procedures as DLL modules?

a) _____

7] Name three ways that an object file containing "library" or common procedures can be attached to a process.

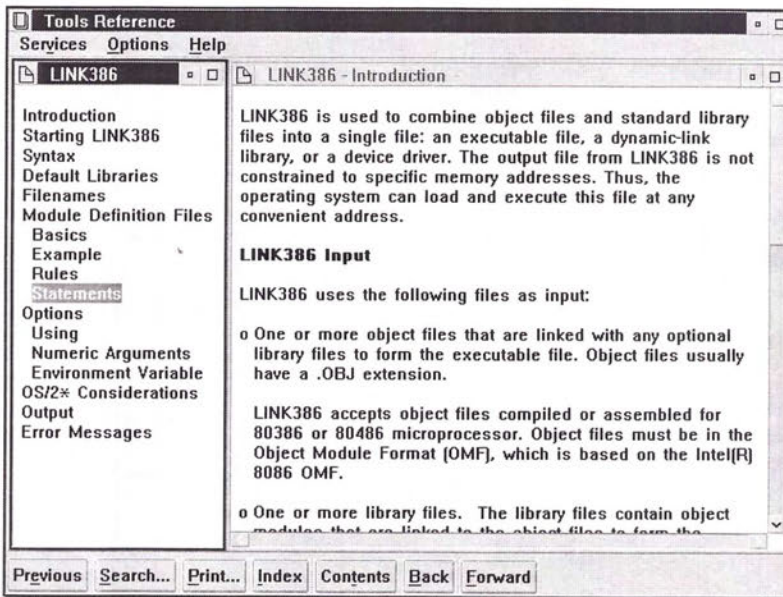
a) _____

8] What are the benefits of run time binding?

a) _____

[] Check your answers on page DLL-28.

Toolkit On Line Reference - DATA statement



DATA ATTRIBUTES:

PRELOAD or LOADONCALL

READONLY or READWRITE

NONE, SINGLE, or MULTIPLE

SHARED or NONSHARED

The data attributes refer to the "automatic data segment" or DGROUP data segment emitted by the compiler. Static data and global variables are placed there.

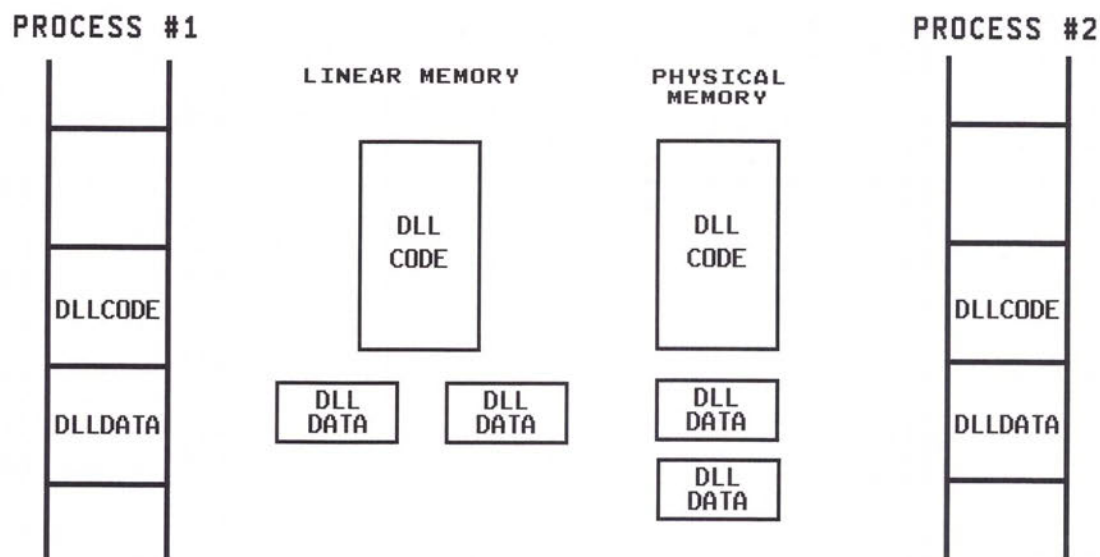
OS/2 2.1 does not honor PRELOAD. Everything is LOADONCALL. OS/2 2.1 uses page demand loading and does the fixups on code every time the code is loaded. Thus code is truly discardable. For READONLY data, OS/2 2.1 will automatically share that data between applications. This is also a way to save RAM.

Classically NONE, SINGLE or MULTIPLE refers to DLL Static Data (DLL Instance Data). To make each attaching application have a new copy of the DLL data, you would specify DATA MULTIPLE.

Classically SHARED or NONSHARED refers to application data. You could purportedly convert you static data into a form of "common" by using DATA SHARED.

We use DATA SINGLE NONSHARED for our DLL Instance Data. This is technically wrong, in that the NONSHARED refers to application data. But it works and it is safe.

DLL Instance Data - DATA MULTIPLE NONSHARED



The diagram above represents linear addresses.

Why MUST the DLL Code be at exactly the same linear address in each process?

Why MUST the DLL Data be at exactly the same linear address in each process?

How is the physical address map different than the linear address map for the DLL and its data segments?

ANSWERS:

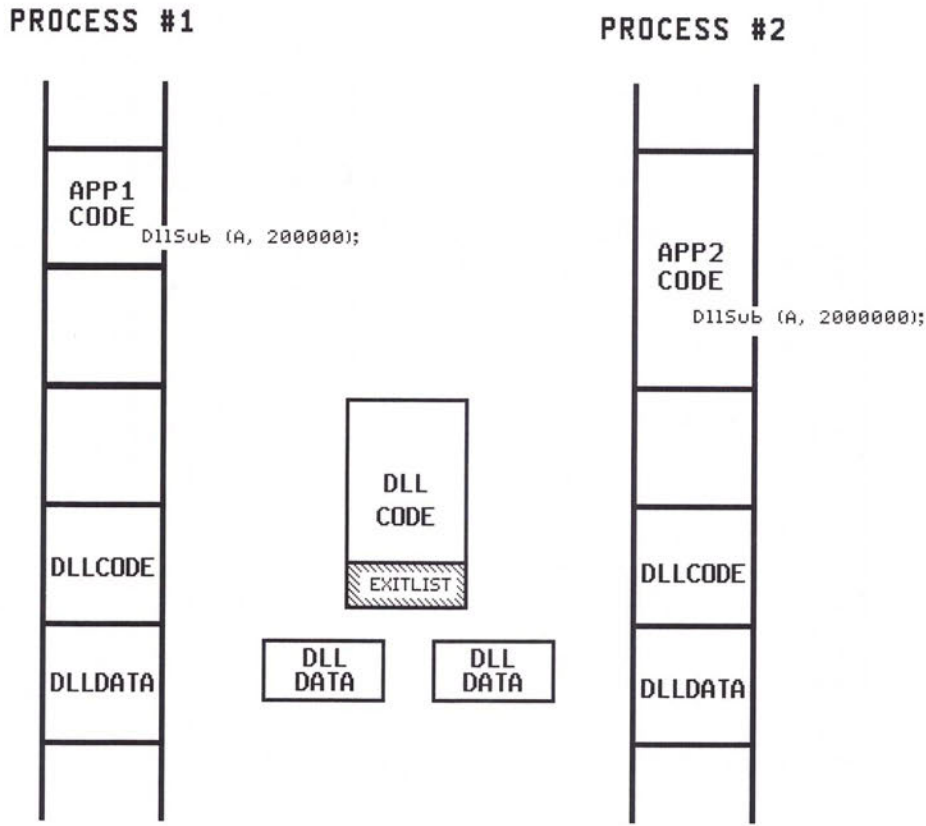
There is only one copy of the DLL code in physical RAM. It is shared between all attaching applications. This is what we call SHARED ADDRESS - SHARED STORAGE memory.

When the DLL code is loaded into memory, there are *fixup* records which are built to allow the DLL to reference itself. The DLL is effectively in ROM (read only memory) once it is loaded in, and thus every process must find the DLL at the same linear address (else the fixups would not work). This is called SHARED ADDRESS - SHARED DATA (or simply shared-shared).

The DLL may reference its data. Thus, the data too must always reside at the same linear address. That is, once the address space has been allocated for the DLL data, then that data must appear there for every instance of the DLL.

The physical address map shows a separate copy of the DLL instance data for each process which attaches. That's why it's called instance data. However, in the linear address map there is but one address assigned to this data. This is called SHARED ADDRESS - PRIVATE DATA (or simply shared-private).

DLL Initialization and Termination



If the DLL must allocate 200,000 or 2,000,000 bytes of storage to keep a working copy of the data array A, how big should it make the static DLL DATA?

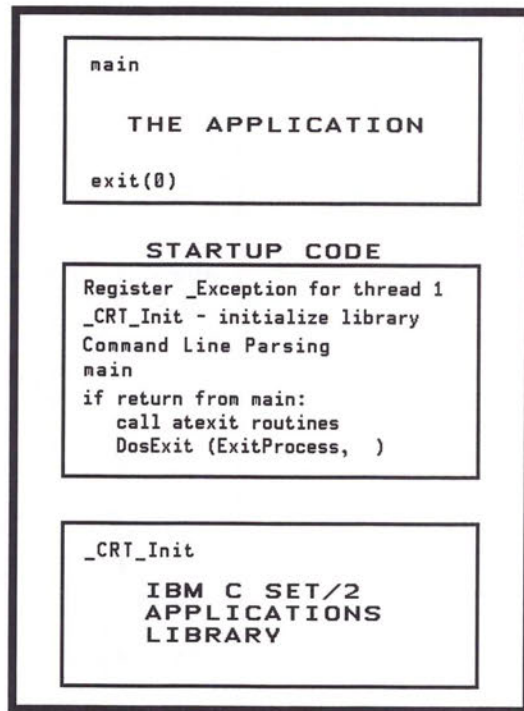
If the DLL allocates resources on behalf of the attaching process, how does it know when to deallocate the resources?

ANSWERS:

There should be no static data array. The array should be allocated dynamically using **DosAllocMem**. The DLL would use static (process specific instance) data to keep track of the buffer address and buffer length, but the memory would be allocated in the private address space of the process by the DLL which would execute a **DosAllocMem** on behalf of the calling process.

The DLL should deallocate the memory if there is a **DosFreeModule**. That is, if the process terminates the DLL early, then the memory should be freed by the DLL (so it will not hang around). If the process simply terminates without freeing the DLL, then OS/2 will free the process' private memory (including the memory which was allocated by the DLL).

_CRT_Init with Static Linking, No User DLLs



The startup code is part of the .EXE file. It calls the `_CRT_Init` entry to initialize the applications library.

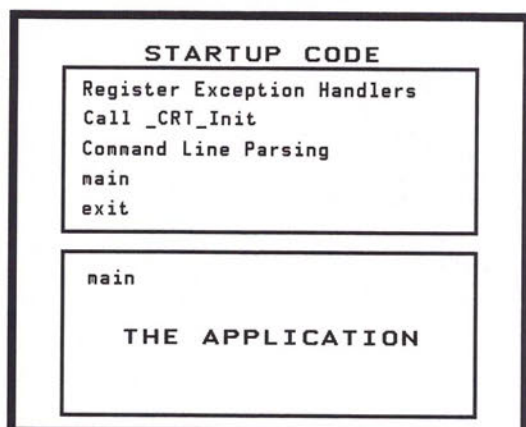
The application should return to the startup code using `exit()` so that the `atexit` routines may be called. `DosExit` will be called by the startup code.

When the application is compiled with `/Gd-` and without the `/Rn` option, the applications library (single or multi-threaded) is statically linked into the .EXE file. Initialization is from the C Startup Code.

`_CRT_Init` does process-specific library initialization. For thread 1 exceptions, the `C` exception handler is registered by the startup code. For all other threads, the `C` exception handler is registered by `_beginthread`. `_beginthread` is roughly equivalent to a `DosCreateThread` and `#pragma` handler.

CRT_Init Application Dynamic, No User DLLs

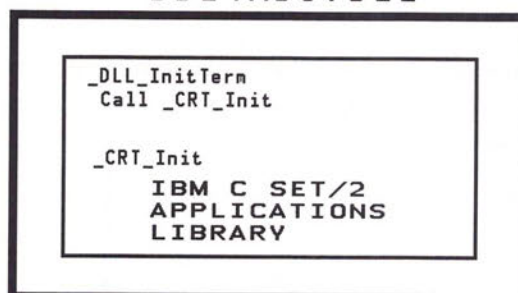
APP.EXE



DDE4MBS.DLL

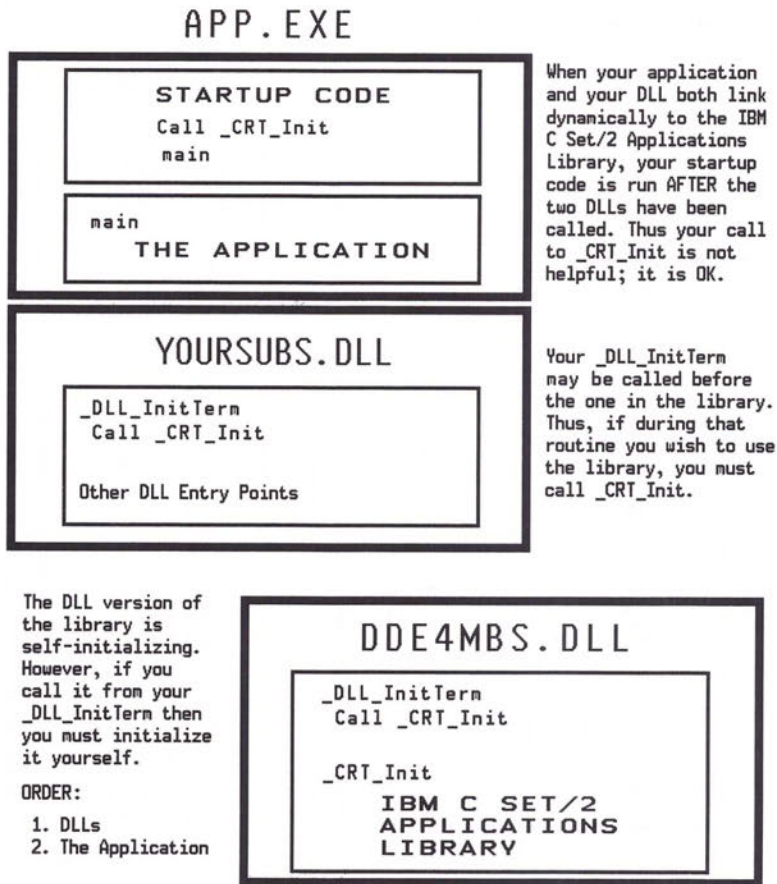
The DLL version of the library is initialized by OS/2 2.1 as it is loaded.

OS/2 2.1 calls _DLL_InitTerm whenever any application attaches the Library.



If the application is dynamically linked to the IBM C Set/2 applications library, the library initializes itself. The call to _CRT_Init from the startup code has no affect, in that the startup code will not call "out of module".

Application Dynamic, User DLLs



When the application AND YOUR DLL are compiled with `/Gd` and without `/Rn` the applications library and YOUR DLL are both dynamically linked into the `.EXE` file. Your DLL uses the dynamic version of the applications library as well.

The DLL version of the C Applications Library is self-initializing: it has its own `_DLL_InitTerm` entry point. However, you must call `_CRT_Init` if you plan to use any of the library functions while your DLL is being initialized. That's because you might end up calling the library before it initializes itself.

The DLL should always handle its own errors. It should not return a signal or an exception to the calling thread but should return an error code. Thus a DLL might want to register its own exception handlers and de-register them on the way out. See `#pragma handler` in appendix E.

If you dynamically link to different versions of the library (say your application links to the single threaded version and your DLL links to the multi-threaded version), you have two *library environments*. You need not worry about this problem if the DLL registers the C exception handler `_Exception` when it is entered, and de-registers it when it leaves. This places the C exception handler for the current library environment at the head of the exception handling list. Again, this is handled for you by `#pragma handler`.

DLL Initialization and Termination

```
#pragma linkage ( _DLL_InitTerm, system)

unsigned long _DLL_InitTerm( unsigned long modhandle, unsigned long flag )
{
    switch( flag )
        /* 0 = initialize, 1 = terminate */
        {
        case 0: /* DLL is being initialized */

            if ( _CRT_init( ) == -1 ) return 0;
            printf ("printf now works in _DLL_InitTerm(0) \n")
            break;

        case 1: /* DLL is being detached */
            break;
        }
    return 1; /* non-zero value returned to continue load */
}
```

```
unsigned long _DLL_InitTerm (      <- 0 = abort process
                                unsigned long modhandle, -> module handle
                                unsigned long flag)      -> 0=load, 1=free
```

You write and include this optional DLL initialization and termination procedure. It will be called by OS/2 so you may execute your own initialization / termination functions.

This routine is NOT optional for statically linked DLLs.

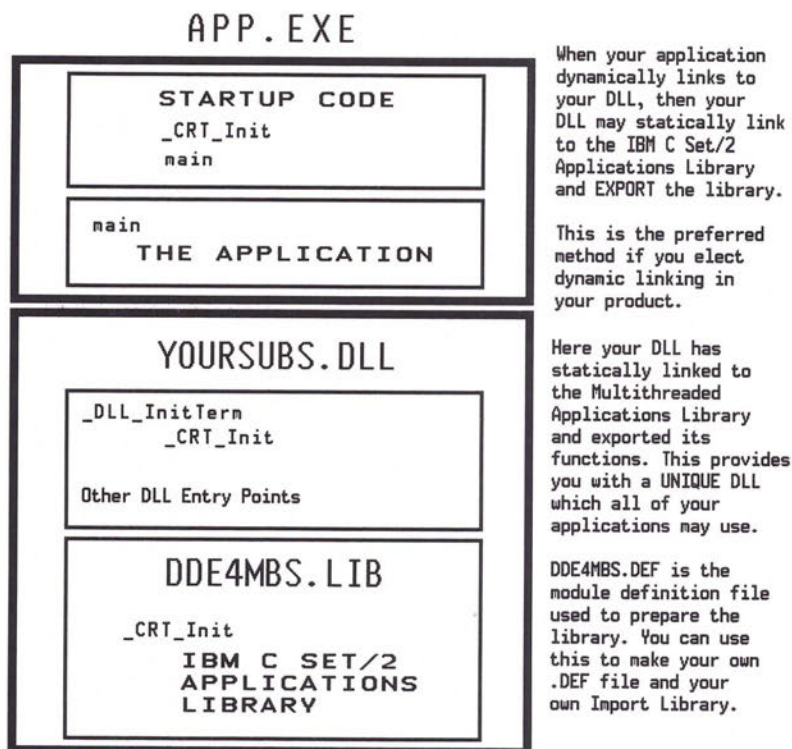
`_DLL_InitTerm` is an entry point name which will be called by OS/2 2.0 based on the initialization and termination options selected in the `LIBRARY` statement of the module definition file. `_DLL_InitTerm` is not specific to IBM C Set/2.

`_CRT_init()` is a function in the IBM C Set/2 Applications Runtime Libraries which initializes the library for thread one. The subsystems library does not require initialization.

In the example above, taken from the lab, if the DLL itself dynamically links to the DLL version of the IBM C Set/2 applications library, and if the DLL wishes to use the library within the initialization case of `_DLL_InitTerm`, then the user must call `_CRT_Init`.

As we will see, the call to `_CRT_Init` is optional when your DLL dynamically links to the applications library, but it is mandatory when your DLL statically links to the applications library. The fact a call to `_CRT_Init` is mandatory does not mean that you have to put one into your DLL when you statically link to the C runtime: the default `_DLL_InitTerm` does that for you. However, if you put in your own `_DLL_InitTerm`, then you *must* have a call to `_CRT_Init` when you statically link to the C runtime.

Shipping the DLL Application Library in your DLL



<Multi-Threaded Application Library> Gm

| | | | |
|-----|----------------|-----------|---------------------|
| [5] | DDE4MBS LIB | Gd- Gm | Static Multithread |
| | + DDE4MBM LIB | Gd- Gm Sm | Migration Library |
| [6] | DDE4MBSI LIB | Gd Gm | Dynamic Multithread |
| | DDE4MBS DLL | | |
| | DDE4MBS DEF | | |
| | DDE4MBSO LIB | | |
| | + DDE4MBMI LIB | Gd Gm Sm | Dynamic Multithread |
| | DDE4MBM DLL | | Migration Library |
| | DDE4MBM DEF | | |

If you want to use dynamic linking in your product, it is recommended that you make your own DLL which statically links in the IBM C Set/2 Applications library. Then you link only to your DLL. This means that your application is not dependent on the existence of the IBM DLLs in your customer's environment, and also that your application can use Dynamic Linking.

Note that in this case, when you make your .EXE, you will use the import library which describes EXPORTS from your DLL. These exports include the C Runtime Library Functions.

The .DEF file in the library may be used to create your EXPORTS statement for your DLL which exports the C Runtime, and the ...O.LIB must be linked into your .EXE or added to your import library, as it is the C Startup Code.

IBM C Set/2 Run Time Libraries

<subsystem library> Rn

| | | | | | | |
|-----|----------|-----|-----|----|---------------------------|--|
| [1] | DDE4NBS | LIB | Gd- | Rn | Static Subsystem Library | |
| [2] | DDE4NBSI | LIB | Gd | Rn | Dynamic Subsystem Library | |
| | DDE4NBS | DLL | | | | |
| | DDE4NBS | DEF | | | | |
| | DDE4NBSO | LIB | | | | |

<Single Threaded Application Library> Gm-

| | | | | | | |
|-----|----------|----------|-----|------------|----------------------|--|
| [3] | DDE4SBS | LIB | Gd- | Gm- | Static Singlethread | |
| | + | DDE4SBM | LIB | Gd- Gm- Sm | Migration Library | |
| [4] | DDE4SBSI | LIB | Gd | Gm- | Dynamic Singlethread | |
| | DDE4SBS | DLL | | | | |
| | DDE4SBS | DEF | | | | |
| | DDE4SBSO | LIB | | | | |
| | + | DDE4SBMI | LIB | Gd Gm- Sm | Dynamic Singlethread | |
| | | DDE4SBM | DLL | | Migration Library | |
| | | DDE4SBM | DEF | | | |

<Multi-Threaded Application Library> Gm

| | | | | | | |
|-----|----------|----------|-----|-----------|---------------------|--|
| [5] | DDE4MBS | LIB | Gd- | Gm | Static Multithread | |
| | + | DDE4MBM | LIB | Gd- Gm Sm | Migration Library | |
| [6] | DDE4MBSI | LIB | Gd | Gm | Dynamic Multithread | |
| | DDE4MBS | DLL | | | | |
| | DDE4MBS | DEF | | | | |
| | DDE4MBSO | LIB | | | | |
| | + | DDE4MBMI | LIB | Gd Gm Sm | Dynamic Multithread | |
| | | DDE4MBM | DLL | | Migration Library | |
| | | DDE4MBM | DEF | | | |

The .DEF file in the library may be used to create your EXPORTS statement for your DLL which exports the C Runtime, and the ...O.LIB must be linked into your .EXE or added to your import library, as it is the C Startup Code.

CRT_init() and Statically Linked DLL

```
#pragma linkage (_DLL_InitTerm, system)

unsigned long _DLL_InitTerm( unsigned long modhandle, unsigned long flag )
{
    switch( flag )
        /* 0 = initialize, 1 = terminate */
        {
        case 0: /* DLL is being initialized */

            if ( _CRT_init( ) == -1 ) return 0;
            printf ("printf now works in _DLL_InitTerm(0) \n")
            break;

        case 1: /* DLL is being detached */
            break;
        }
    return 1; /* non-zero value returned to continue load */
}
```

```
int _CRT_init(void)          -> -1 = failure
```

This routine initializes the C Runtime. It is not required for the subsystem library (which has no initialization).

This routine must be used (1) if your DLL dynamically links to the applications library AND you make library calls during your initialization; or (2) when your DLL statically links to the IBM C Set/2 applications library.

A statically linked DLL must initialize the C Runtime by calling `_CRT_init`. While those library modules which are DLLs can initialize themselves via initialization entry points, and library routines which are statically attached to the .EXE file can be initialized by the C startup code associated with the `main()` procedure, library routines statically bound to a DLL must be initialized by you.

LIBRARY MYDLL INITGLOBAL TERMGLOBAL

Syntax: LIBRARY [libraryname] [initialization] [termination]

This statement identifies the executable file as a dynamic-link library and optionally defines the name and library module initialization required.

If <libraryname> is given, it becomes the name of the library as it is known by OS/2. This name can be any valid file name. If <libraryname> is not given, the name of the executable file - with the extension removed - becomes the name of the library.

If <initialization> is given, it defines the library initialization required and can be one of the values below. If omitted, <initialization> defaults to INITGLOBAL.

INITGLOBAL

The library initialization routine is called only when the library module is initially loaded into memory. Using this keyword without a termination flag implies TERMGLOBAL for DLLs with 32-bit entry points.

INITINSTANCE

The library initialization routine is called each time a new process gains access to the library. Using this keyword without a termination flag implies TERMINSTANCE for DLLs with 32-bit entry points.

If <termination> is given, it defines the library termination required and can be one of the values below. If omitted, <initialization> defaults to TERMGLOBAL. The termination flag can only apply to DLLs with 32-bit entry points.

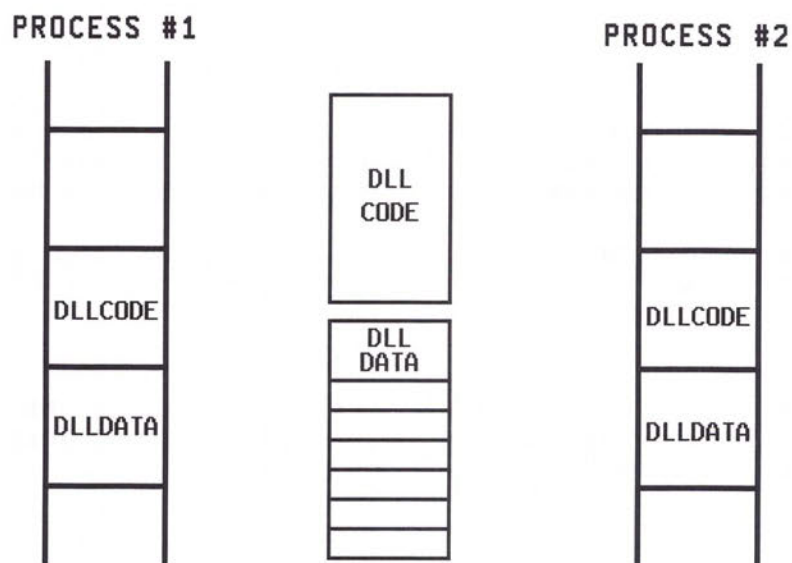
TERMGLOBAL

The library termination routine is called only when the library module is unloaded from memory. Using this keyword without an initialization flag implies INITGLOBAL.

TERMINSTANCE

The library termination routine is called each time a process relinquishes access to the library. Using this keyword without an initialization flag implies INITINSTANCE.

DLL with DATA SINGLE SHARED



The diagram above represents linear addresses.

How is the physical address map different than the linear address map for the DLL and its data segments?

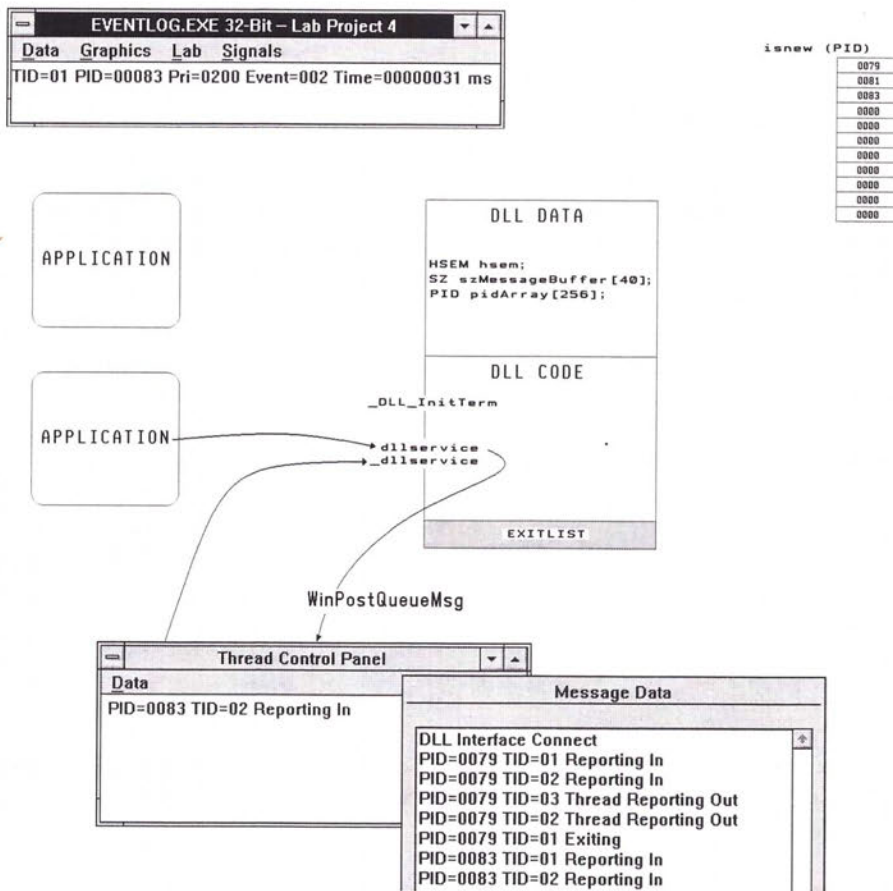
How can we make sure that the DLL and its data is not discarded, so that the DLL can manage a hardware or software resource?

How can we protect the shared DLL data from accidental destruction by any one of the applications which attaches the DLL?

Using a DLL to manage a hardware or software resource is an alternative strategy to using a process to manage the resource. The benefit of a DLL is that whenever an attaching process is running, the DLL (and its shared data) is in the context of that process: local calls (no IPC) may be used to communicate between the DLL and the application. The DLL effectively "carries" its data to whichever application is attaching it and running.

A subsystem can be comprised of a DLL, a detached process, a Device Driver, and a piece of hardware. The DLL provides the top level subsystem interface.

Using a DLL to Manage a Resource



Unlike EVENTLOG which was a program which, due to its use of shared memory, only communicated with one application, the DLL here can be attached by any number of applications. When attached and called at entry point `_dllservice`, the DLL recognizes that it is being called by the "thread controller" which is the unique resource. The DLL passes back a message.

Any application which attaches the DLL calls the entry point `_dllservice`. If this is the first time that particular PID has called the entry, routine `isnew` returns TRUE and adds the PID to its array of attached PIDs. A message is put into the message buffer and IPC using `WinPostQueueMessage` is used to notify the thread controller (PM application) that a new process has attached. Using an older technique, `DosExitList` rather than `_DLL_InitTerm` was used to detect the termination of the process. Thus if `isnew` returns TRUE, the EXITLIST is registered.

DISCUSSION:

If a semaphore is created on the `_dllservice` entry, what must be done so that semaphore may be used when applications call the `_dllservice` entry?

Answers to Questions

< page DLL-13 >

- 1] When an application executes **DosAllocSharedMem** linear address space is used in which processes? When will that linear address space be freed?
 - a] Linear address space is reserved in all processes which are currently running, and all future processes which will be loaded. The linear address space is freed when the usage count (or "reference" count) on the shared object goes to zero.

- 2] When an application attaches a DLL, linear address space is used in which processes? When will that linear address space be freed?
 - a] The answer is the same as the answer to question #1.

- 3] Under what circumstances will code be shared between two processes which use static linking? What do you have to do to get this to happen?
 - a] If one process is loaded multiple times, then the code is shared between them. You don't have to do anything: OS/2 does this for you.

- 4] Under OS/2 1.3, how could you share statically bound data segments between processes? Why would you want to do this? What is an easier (and perhaps safer) way to share data?
 - a] To share statically declared data such as arrays, you could specify in the module definition file that that particular array should be shared. This is a kind of "common" between processes. An easier way, of course, is to dynamically allocate shared memory at run time. It is safer in that some of these module definition file options may not work...

- 5] How can you get .EXE files with different names and *some* common code to share the code?
 - a] Put the code in a DLL.

- 6] What is the primary benefit of writing "library" or common procedures as DLL modules?
 - a] Space is saved in RAM when *different* .EXE files want to share the same code. Space is also saved on disk.

7] Name three ways that an object file containing "library" or common procedures can be attached to a process.

a] You can statically link it. You may put the code in a DLL and attach the DLL by reference (import the DLL), or you may put the code in a DLL and load it at run time using **DosLoadModule**. The two DLL techniques are called load time linking (binding) and run time linking (binding).

8] What are the benefits of run time binding?

a] You may have many DLLs and only attach those you need. You may use **DosFreeModule** to tell OS/2 that you no longer need the DLL. If your DLL is not available, your program will still load and you can deal with the error yourself. The DLL need not be in the LIBPATH. See the "Application Design Guide".

d] By using child processes and by using run time binding, you may explicitly tell OS/2 when you are done with a code resource. This simplifies the job of the memory manager, in that it does not have to "guess" based on LRU (least recently used) algorithms when a piece of code is a good candidate to discard in a memory overcommit situation.

OS/2 2.1 for Software Developers

Appendix E

Exceptions and Handlers

Some material in this chapter
courtesy IBM Toronto Lab

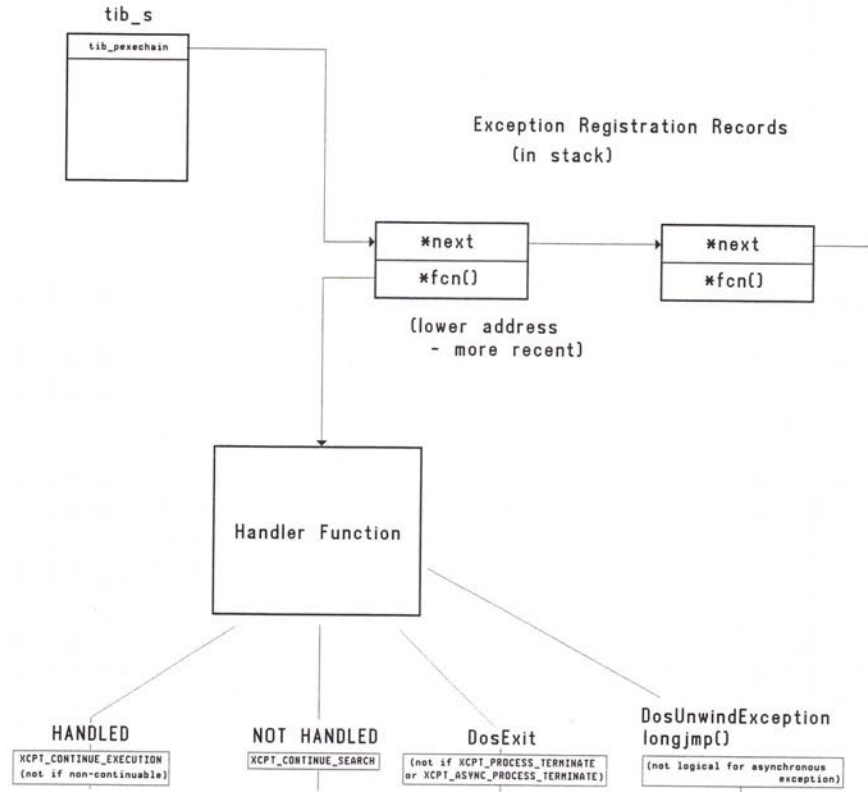
by Charles Chernack

2 Table of Contents

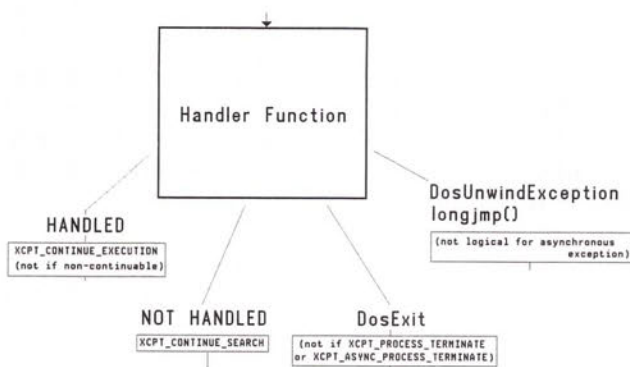
| | |
|---|------|
| Chapter E - The OS/2 2.1 Desktop | E-1 |
| Exception Registration Records and Handlers | E-3 |
| The Handler | E-4 |
| Using setjmp and longjmp | E-5 |
| function chkptr (sample) | E-6 |
| setjmp function | E-7 |
| longjmp function | E-8 |
| The C _Exception Handler | E-9 |
| #pragma handler | E-10 |
| Writing an Exception Handler | E-11 |
| Using #pragma map to register your handler | E-14 |

Exception handlers are registered by placing exception registration records in the thread stack. The `tib_s` structure contains a ULONG pointer to the most recent exception registration record, which is at the lowest address in the stack.

THE CHAIN OF HANDLERS: An exception registration record is made up of two ULONGS, a forward pointer to the next registration record and a pointer to the handler. If an exception occurs in the context of this thread, the first handler in the chain is called. If that handler returns `XCPT_CONTINUE_SEARCH`, then the next handler is called, and so on.



The Handler



XCPT_CONTINUE_EXECUTION: If the handler "fixes" the problem, then it returns

XCPT_CONTINUE_EXECUTION. For example, if a handler in thread one decides to ignore **XCPT_SIGNAL_KILLPROC** (see page 13-6 of the OS/2 2.0 Programming Guide), it can return **XCPT_CONTINUE_EXECUTION**.

XCPT_CONTINUE_SEARCH: If the handler does not deal with this kind of exception, then it can pass the buck to the next handler in the chain by returning **XCPT_CONTINUE_SEARCH**. For example, if you write a handler to handle page faults, and the exception is not a page fault, you would return **XCPT_CONTINUE_SEARCH**.

DosExit (EXIT_THREAD or EXIT_PROCESS...): The exception handler, or a signal handler called by the exception handler, may decide to terminate the thread or process based on the exception. In that case, there is no return from the handler.

Note that the handler must not do a **DosExit** if the exception is a **...PROCESS_TERMINATE** exception. Whenever a process terminates itself, or a process is terminated by an external request, the exception handler for each thread is called with a **XCPT_PROCESS_TERMINATE** (terminating itself) or **XCPT_ASYNC_PROCESS_TERMINATE** (external termination) exception. If the handler were to do a **DosExit** in this case, there would be a loop!

longjmp() or DosUnwindException: The handler may **longjmp** back to the point prior to the problem. We have an example of that following. In that case, there is no return from the handler and the process continues. This is typically used for segment violation. This is not logically useful for an asynchronous (externally caused) exception. **DosUnwindException** is not useful for C programmers. It is kind of a self-generated **longjmp** without a corresponding **setjmp**, and can best be done at the assembly level by programmers who work at that level.

Using setjmp and longjmp

```
extern int chkptr (void * ptr,          /* pointer to storage to check */
                  int size);          /* number of bytes to check */

#include <stdlib.h>
#include <stdio.h>

int main(void) {
    char * x = malloc (60000);
    printf ("Trying chkptr()\n");
    printf ("Chkptr reports %d available bytes\n", chkptr(x,120000));
    return 0;
}
```

This example is the main program for the setjmp/longjmp signal handling example following, and the setjmp/longjmp exception handling example following. Here we malloc 60000 bytes and then call procedure chkptr which will try to access 120000 bytes of the array. Procedure chkptr returns the size of the array by accessing consecutive elements until it gets a memory protect violation.

Two ways of surviving the memory protect violation without terminating the offending thread are demonstrated: setjmp/longjmp using a signal handler and setjmp/longjmp using an exception handler.

Using setjmp and longjmp

```
#include <signal.h> // for the signal function
#include <setjmp.h> // for the setjmp and longjmp functions
#include <stdio.h> // for the printf call

static void myhandler (int sig); // the signal handler prototype
static jmp_buf jbuf; // save area for longjmp

int chkptr(void * ptr, // pointer to storage to check
           int size) // number of bytes to check
{
    void (* oldsig)(int); // where to save the old signal handler
    volatile char c; // volatile to insure access occurs
    int valid = 0; // count of valid bytes
    char * p = ptr; // to satisfy the type checking for p++

    oldsig = signal(SIGSEGV, myhandler); // set the signal handler

    if (!setjmp(jbuf)) { // provide a point for the signal handler
                          // to return to

        while (size--) // scan the storage
        {
            c = *p++; // check the storage
            valid++; // then bump the counter
        }

        signal(SIGSEGV, oldsig); // reset the signal handler
        return valid; // return number of valid bytes
    }

    static void myhandler (int sig) {
        printf("Invalid address\n"); // restart the function at the setjmp() call
        longjmp(jbuf, 1); // without restarting the while() loop
    }
}
```

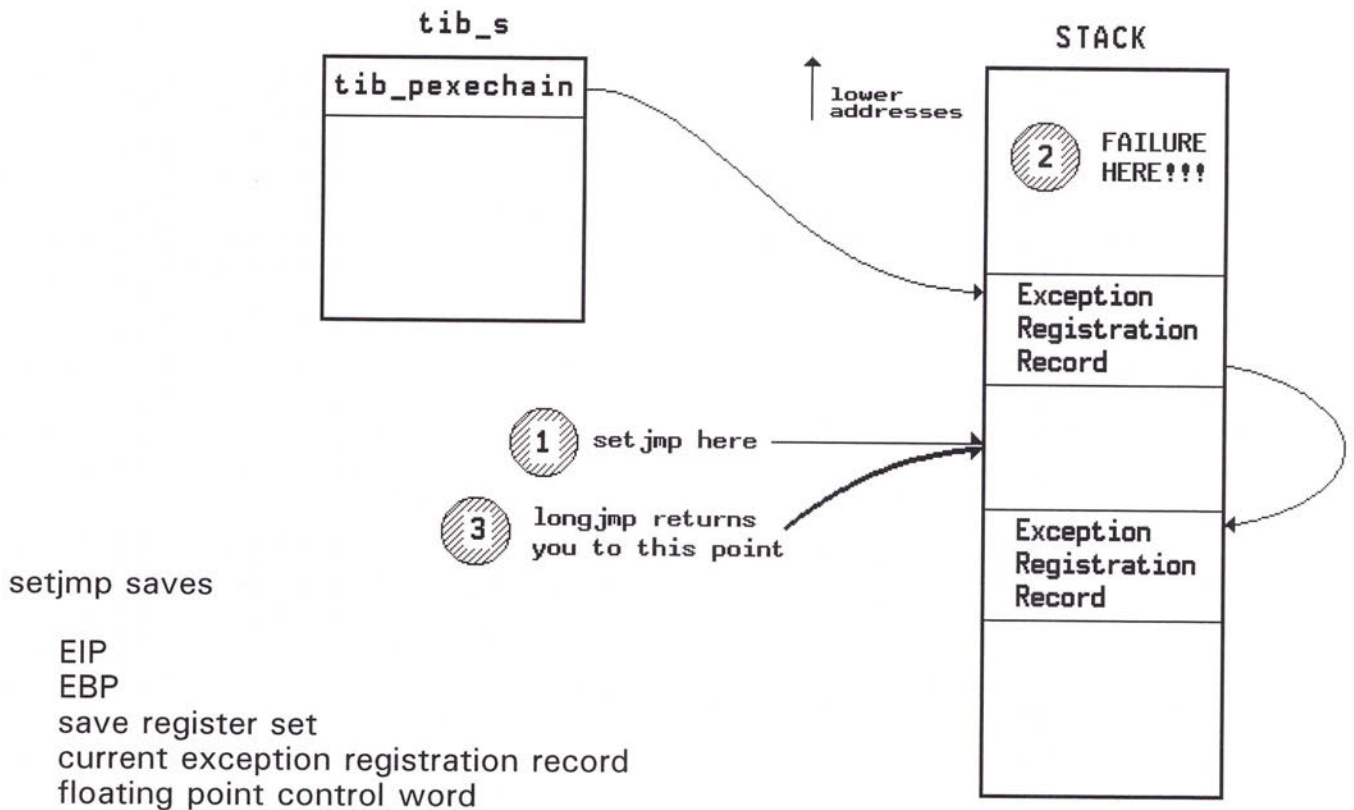
Procedure `chkptr` registers a signal handler for `SIGSEGV`. It saves the previous content of the threads signal handling table for `SIGSEGV`, and upon return resets the threads signal handling table.

`jbuf` is a static storage area which contains information saved by `setjmp` and used by `longjmp`. Because we have but one `jbuf`, this version of `chkptr` is NOT reentrant.

The first call to `setjmp` saves the machine state at this point in the program, and returns `FALSE` so that the while loop is entered. To prevent the compiler from optimizing out the code `c = *p++`, `c` is declared `volatile`. That tells the compiler NOT to carry "c" in a register and to execute the memory reference.

When the memory protection violation occurs, our signal handler is called. It prints a message and does a `longjmp` back to the most recent `setjmp`. Since a 1 is passed, `setjmp` returns `TRUE` and the while loop is skipped. Procedure `chkptr` resets the state of the `SIGSEGV` signal and returns the size of the array.

int setjmp (jmp_buf env)



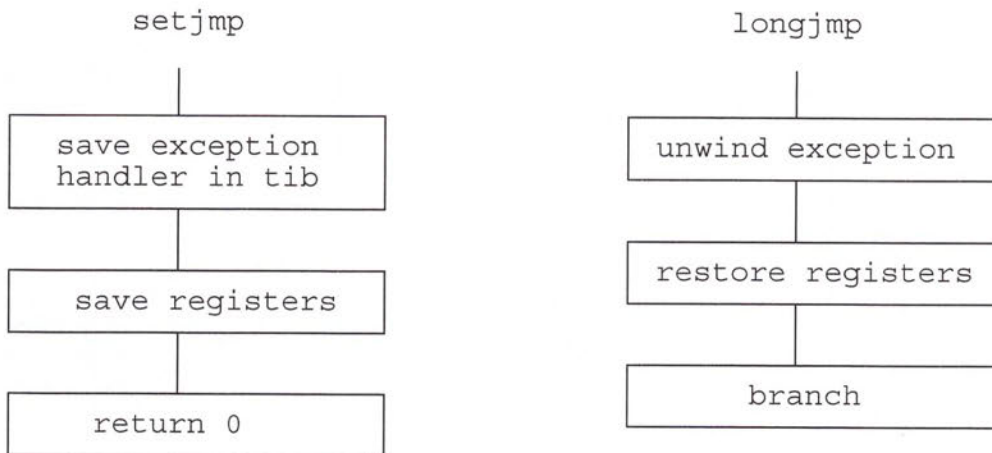
Always use `jmp_buf` to define the `setjmp` buffer.

The `setjmp` function saves a stack environment that can subsequently be restored by `longjmp`. The `setjmp` and `longjmp` functions provide a way to perform a nonlocal goto.

A call to `setjmp` causes it to save the current stack environment. A subsequent call to `longjmp` restores the saved environment and returns control to a point corresponding to the `setjmp` call. The values of all variables (except register variables) accessible to the function receiving control contain the values they had when `longjmp` was called. The values of register variables are unpredictable. Nonvolatile auto variables that are changed between calls to `setjmp` and `longjmp` are also unpredictable.

The `setjmp` function returns the value 0 after saving the stack environment. If `setjmp` returns as a result of a `longjmp` call, it returns the value argument of `longjmp`, or 1 if the value argument of `longjmp` is 0. There is no error-return value.

void longjmp (jmp_buf env, int value)



```
if (!setjmp(jbuf)) {  
    while (size--)  
    {  
        c = *p++;  
        valid++;  
    }  
}
```

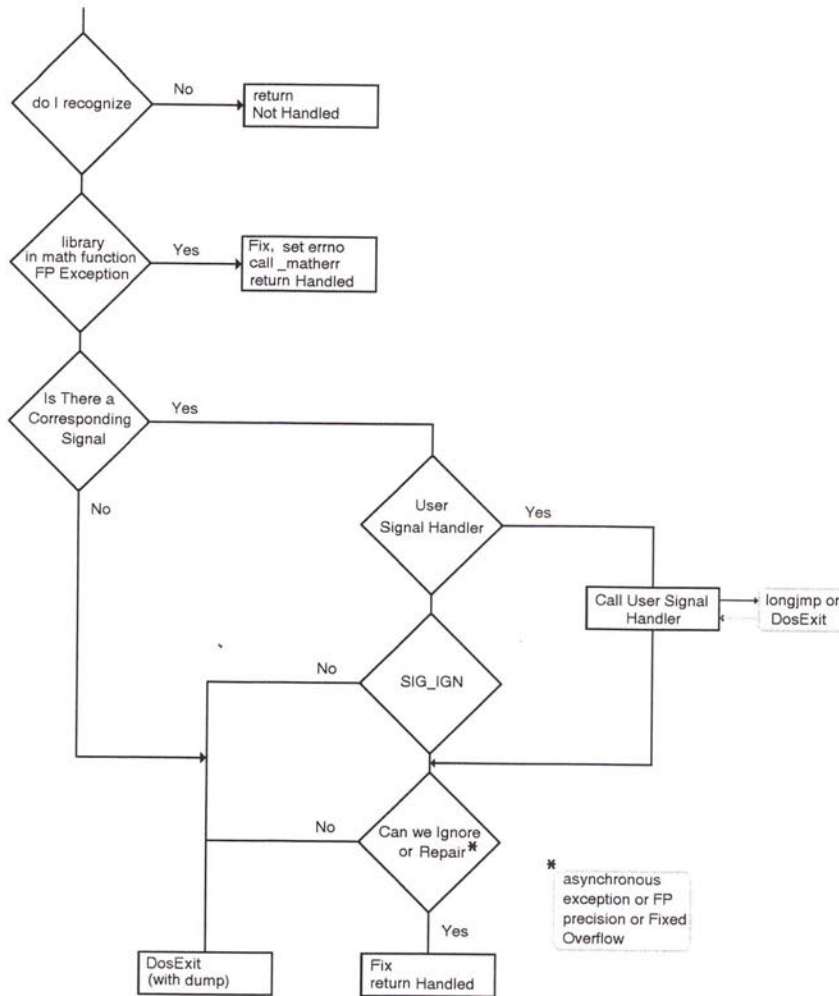
```
static void myhandler (int sig) {  
    printf("Invalid address\n");  
    longjmp(jbuf, 1);  
}
```

The longjmp function restores a stack environment previously saved by setjmp. The setjmp and longjmp functions provide a way to perform a nonlocal goto.

The value argument passed to longjmp must be nonzero. If you give a zero argument for value, longjmp substitutes a 1 in its place.

The longjmp function does not use the normal function call and return mechanisms; it has no return value.

The C _Exception Handler

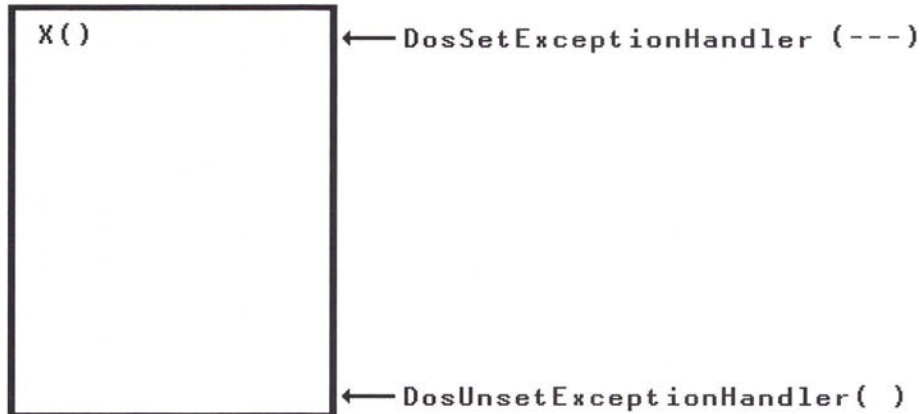


The matherr function allows users to process errors generated by the functions in the math library. The math functions call matherr whenever they detect an error. The matherr function supplied with the C Set/2 library performs no error handling and returns 0 to the calling function. You can provide a different definition of the matherr function to carry out special error handling.

The _Exception handler is different in IBM C Set ++. The diagram above is for the IBM C Set/2 compiler runtime. for IBM C Set ++ the DosExit is not done (lower left box). This causes return back to the operating system which then takes the default action. The change was made because the DosExit (with dump) made Presentation Manager applications "evaporate" on a fatal fault. By returning to the operating system, the hard error screen comes up (unless suppressed via a DosError call or AUTOFAIL in CONFIG.SYS).

#pragma handler - register _Exception for function

```
>>#pragma handler (function) <<
```



The #pragma handler directive generates the code at compile time to install the C Set/2 exception handler, _Exception, before starting execution of the function. Code to remove the exception handler at function exit is also generated.

You may use this directive whenever you change library environments or enter a user-created DLL. The function is the name of the function for which the exception handler is to be registered. This function should be declared before you use it in this directive.

If you are using the subsystem libraries, the _Exception function is not provided. To use the #pragma handler directive in a subsystem, you must provide your own exception handler named _Exception.

Writing an Exception Handler (1 of 3)

```
#define INCL_DOS
#define INCL_NOPMAPI          /* no PM api */
#include <os2.h>              /* for the doscall */
#include <stdlib.h>           /* for the signal function */
#include <setjmp.h>           /* for the setjmp and longjmp functions */
#include <stdio.h>           /* for the printf call */

void * tss_array[100];      /* array for 100 thread specific pointers */

static TID getTid(void);    /* returns our TID */

APIRET APIENTRY MyExceptionHandler(EXCEPTIONREPORTRECORD *,
                                    EXCEPTIONREGISTRATIONRECORD *,
                                    CONTEXTRECORD *,
                                    PVOID);

#pragma map(_Exception,"MyExceptionHandler")
#pragma handler(chkptr)

int chkptr(void * ptr,      /* pointer to storage to check */
           int size)      /* number of bytes to check */
{
    volatile char c;      /* volatile to insure access occurs */
    int valid = 0;        /* count of valid bytes */
    char * p = ptr;      /* to satisfy the type checking for p++ */
    jmp_buf jbuf;        /* the jump buffer moves to automatic storage */
                        /* so that it is unique to this thread */

    /* create a thread specific jmp_buf */
    tss_array[getTid()] = (void *)jbuf;

    if (!setjmp(jbuf)) {  /* provide a point for the signal handler */
                        /* to return to */

        while (size--)   /* scan the storage */
        {
            c = *p++;    /* check the storage */
            valid++;     /* then bump the counter */
        }
    }

    return valid;        /* return number of valid bytes */
}
```

Writing an Exception Handler (2 of 3)

```
APIRET APIENTRY MyExceptionHandler(EXCEPTIONREPORTRECORD * report_rec,
                                   EXCEPTIONREGISTRATIONRECORD * register_rec,
                                   CONTEXTRECORD * context_rec,
                                   PVOID dummy)
{
    /* check the exception flags */
    if (EH_EXIT_UNWIND & report_rec->fHandlerFlags)        /* exiting */
        return XCPT_CONTINUE_SEARCH;

    if (EH_UNWINDING & report_rec->fHandlerFlags)          /* unwinding */
        return XCPT_CONTINUE_SEARCH;

    if (EH_NESTED_CALL & report_rec->fHandlerFlags)       /* nested exceptions */
        return XCPT_CONTINUE_SEARCH;

    /* determine what the exception is */
    if (report_rec->ExceptionNum == XCPT_ACCESS_VIOLATION) {
        /* this is the only one we expect */

        printf("Detected invalid storage address\n");
        longjmp((int *)tss_array[getTid()], 1);           /* restart the function at setjmp() call */
                                                         /* without restarting the while() loop */

    } /* endif */

    return XCPT_CONTINUE_SEARCH;                          /* we don't handle this exception */
}

static TID getTid(void)                                   /* return TID of current thread */
{
    TIB *ptib;                                           /* pointer to a thread information block */
    PIB *ppib;                                           /* pointer to a process information block */

    DosGetInfoBlocks( &ptib, &ppib );
    return (ptib->tib_ptib2->tib2_ultid);
}
```


Writing an Exception Handler (3 of 3)

The version of `chkptr` shown on the previous pages is reentrant because it declares `jbuf` as an automatic variable. In order for the exception handler to do the `longjmp` through the thread specific `jbuf`, an array of 100 pointers (supporting 100 threads) is used: the exception handler cannot reference the actual `jbuf` because of scoping.

Since we are writing our own exception handler, we must provide the function prototype. The `EXCEPTIONREPORTRECORD` is what we use. The `EXCEPTIONREGISTRATIONRECORD` is a pointer to our exception registration record (no need to use this), and the `CONTEXTRECORD` contains the CPU context. Do not modify the `CONTEXTRECORD`. Assembler programmers may find the contents interesting...

The `pragma map` combined with the `pragma handler` represent a (possibly undocumented) method to get the C Set/2 Compiler to register and de-register your handler for you when you enter and leave function `chkptr`. This concept of putting your own special-purpose limited-function handler in at the front of the list of the duration of one procedure only is typical.

Inside the exception handler, the prudent policy is to leave with `XCPT_CONTINUE_SEARCH` if you do not have anything specific to do regarding this exception, or if you do not recognize it or wish to process it.

We check the EH flags which are set to indicate exception attributes first, as they might be set along with `XCPT_ACCESS VIOLATION`. These EH flags are described in the [On Line Programming Guide and Reference](#).

If we pass the EH "filter" we check for the specific exception we wish to process. If we find it, we do a `longjmp` which means that we never return from the exception. OS/2 2.0 had "interrupted" our thread to send us the exception, and we simply continue in our thread.

If we do not find the specific exception we want, we return `XCPT_CONTINUE_SEARCH`.

```
>>#pragma map(—identifier—,—"name"—)><
```

The `#pragma map` directive associates an external name (name) with a C name (identifier).

When you use this directive, the identifier that appears in the source file is not visible in the object file. The external name will be used to represent the object in the object file.

The map name is an external name and must not be used in the source file to reference the object. If you try to use the map name in the source file, to access the corresponding C object, the compiler will treat the map name as a new identifier.

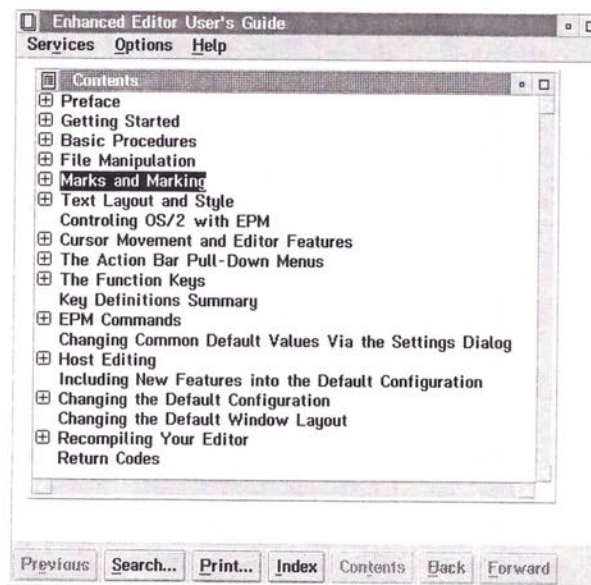
```
#pragma map( Exception, "MyExceptionHandler")  
#pragma handler(chkptr)
```

In this example, `#pragma handler` causes the IBM C Set/2 Compiler to register and deregister the C exception handler `_Exception` when function `chkptr` is entered and exited. The use of `#pragma map` gets the compiler to emit code to register `MyExceptionHandler` instead.

OS/2 2.1 for Software Developers

Appendix EDIT

Text Editors E.EXE and EPM.EXE



VIEW EPMUSERS

by Charles R. Chernack

2 Table of Contents

| | |
|---|---------|
| Appendix EDIT - Using E.EXE and EPM.EXE | EDIT-1 |
| Using the PC DOS 6.3 E.EXE Full Screen Editor . | EDIT-3 |
| Introduction | EDIT-3 |
| Using the Editor | EDIT-4 |
| The Editor Help File | EDIT-5 |
| Using the EPM 6.0 Presentation Manager Editor . | EDIT-11 |
| Introduction | EDIT-12 |
| Setting Preferences | EDIT-12 |
| Fonts | EDIT-13 |
| Enter Key(s) | EDIT-13 |
| Colors | EDIT-13 |
| Using the Command [Line] Dialog | EDIT-13 |
| Go to a Line Number | EDIT-14 |
| Options - List Ring | EDIT-14 |
| File Menu - Accelerators, Open vs Add File | EDIT-15 |
| Search | EDIT-15 |
| Edit - Line and Block Marking, Copying, etc | EDIT-16 |
| The Editor Help File | EDIT-17 |
| Accelerator Key Definitions | EDIT-17 |
| Command Line Commands | EDIT-18 |

NOTES:

Significant material courtesy of IBM Corporation, E Editor and EPM Editor Help files.

Using the E Editor

INVOKING THE EDITOR - EDITING MULTIPLE FILES

You can invoke the editor using E, EFS or EMAX. You may use wild cards in the invocation:

```
E M*.C M*.ERR          <- edit main.c and main.err in our labs
```

You may go around the ring of files using F10. You can also bring in additional files once the editor is started by going to the command line and typing E <filename>. When you have quit or saved all of the files, the editor terminates.

MOVING OR COPYING TEXT LINES

You can mark a line of text using ALT+L. To mark many lines, move the cursor and enter ALT+L again. Once line(s) are marked, you can copy the line(s) using ALT+C or move them using ALT+M. ALT+U unmarks the lines. ALT+Y moves the cursor to the marked area. All of these rules remain the same if you are working with multiple files in the ring.

DELETING, SPLITTING, COMBINING LINES

CTRL+Backspace will delete the line the cursor is on. ALT+D will delete the marked lines. ALT+S will split a line at the cursor. ALT+J will join lines. CTRL+E is a delete right -- deleting all of the character from the cursor to the end of the line.

SHIFTING TEXT TO THE LEFT, TO THE RIGHT

If lines are marked, ALT+7 will shift the text to the left. ALT+8 will shift the text to the right.

INCLUDING ONE FILE WITHIN ANOTHER

On the command line, use GET filename. The file specified will be copied into the file you are currently editing.

BLOCK MARKING

The editor has convenient block marking commands. They allow you to establish a marked rectangle on the screen, and then copy, move, fill, clear, or shift the marked area. The commands are ALT+B to mark each corner, ALT+F to fill (followed by the fill character), ALT+D to delete the block, ALT+U to unmark the block. ALT+O will overlay the block at the current cursor position.

COMMAND LINE RECALL

When on the command line, the up and down arrow will bring back previous commands.

Using the E Editor (continued)

SEARCHING FOR TEXT, CHANGING TEST

c/old/new/*m will change all occurrences of "old" to "new" in the marked area. If you leave off the "m", then all occurrences will change. The changes are from the current cursor position forward.

f/findtext/- will search for "findtext" backwards. If you leave out the "-" it will search forward. CTRL+F will search again.

CHANGING THE NAME OF A FILE

Using the DOS command on the command line, you can shell out to a command prompt. You may use the NAME command (or N) to give the file you are currently editing a new filename.

GOING TO A LINE NUMBER - DEALING WITH COMPILER ERROR MESSAGES

By default the IBM C Set Compiler routes its errors to stdout. Our lab makefiles redirect the errors to a file with the extension .ERR.

When you get an error with the compiler, you can go to that specific line number by going to the command line and typing the line number and hitting enter. Since we generally route our error messages to a .ERR file in the lab makefiles, I have provided files X.CMD, Y.CMD and M.CMD for use with the E editor (you may modify them for EPM). Y.CMD says E L*.C L*.ERR which would bring up the .C and .ERR file in the ring of E.EXE. Note that the IBM C Set compiler provides error messages which specify the line number and column number.

SYNTAX EXPANSION - EXPAND ON

If the file you are editing has the extension .c (such as lab1.c), and you type EXPAND ON on the command line, then statements such as FOR and IF will be expanded automatically. That is, if you type **for** you will see:

```
for ( ; ; ) {  
  /* endfor */
```

SORT

The SORT command on the command line will sort the entire file, or if there are marked lines, it will sort the marked area.

FINDING DOCUMENTATION

See the IBM PC DOS 6.1 or PC DOS 6.3 manual.

File EHELP.HLP (1/5)

| | | | |
|---------------|------------------------------------|--|---------|
| E Editor Help | Directory of Help Panels | | |
| Page 1: | Function Keys and Alt+Keys | (F1 - F10) (Alt+F1 - Alt+F10) | |
| Page 2: | Shift+Keys and Ctrl+ Keys | (Shift+F1 - Shift+F10) (Ctrl+F1 - Ctrl+F10) | |
| Page 3: | Alt+ Summary | (Alt - Alt+O) | |
| Page 4: | Alt+Summary and Ctrl+ Summary | (Alt+P - Alt+=) (Ctrl+Enter - Ctrl+Backspace) | |
| Page 5: | Ctrl+ Summary | (Ctrl - Ctrl+Z) | |
| Page 6-9: | Command Summary | | |
| Page 10: | Edit Commands | | |
| Page 11: | Termination Commands | | |
| Page 12: | Cursor Movement Controls | | |
| Page 13: | Copying, Moving, and Deleting text | | |
| Page Down | | | F3=Exit |

| | | |
|-------------------|--|------------------|
| E Editor Help | Function Keys and Alt+ Keys (F1 - F10) | Page 1 |
| | | Alt + |
| ESC | Moves cursor between the edit area and the command line | |
| F1 | Display help text | Box characters |
| F2 | Save file and continue | |
| F3 | Quit without saving file | |
| F4 | Save file and quit | |
| F5 | | |
| F6 | Show draw options | |
| F7 | Change filename | Shift mark left |
| F8 | Edit new file | Shift mark right |
| F9 | Undo current line | |
| F10 | Next file | Previous file |
| Page Down Page Up | | F3=Exit |

| | | |
|-------------------|----------------------------------|-----------------------------|
| E Editor Help | Shift+ and Ctrl+ Keys (F1 - F10) | Page 2 |
| | | |
| | Shift + | Ctrl + |
| F1 | Scroll left | Uppercase word |
| F2 | Scroll right | Lowercase word |
| F3 | Scroll down | Uppercase mark |
| F4 | Scroll Up | Lowercase mark |
| F5 | Center line vertical | Cursor to beginning of word |
| F6 | | Cursor to end of word |
| F7 | | |
| F8 | | |
| F9 | | |
| F10 | | |
| Page Down Page Up | | F3=Exit |

File EHELP.HLP (2/5)

| E Editor Help | Alt+ Keys (Alt - Alt+R) | Page 3 |
|--|-------------------------|---------|
| Alt : change function line descriptions Alt+A : ADJUST marked area, blank old Alt+B : mark BLOCK Alt+C : COPY mark Alt+D : DELETE marked area Alt+E : cursor to END of marked area Alt+F : FILL marked area Alt+J : JOIN (with following line) Alt+L : mark LINE Alt+M : MOVE marked area Alt+N : key in file NAME at cursor Alt+O : OVERLAY blocked area Alt+P : reformat following PARAGRAPH Alt+R : REFLOW marked area | | |
| Page Down | Page Up | F3=Exit |

| E Editor Help | Alt+ Keys (Alt+S - Alt+=) and Ctrl Keys | Page 4 |
|--|---|---------|
| Alt+S : SPLIT line at cursor Alt+T : CENTER text in marked block Alt+U : UNMARK Alt+W : mark WORD Alt+X : ESCAPE (allow special characters) Alt+Y : cursor to beginning of mark Alt+Z : mark, character mode Alt+l : Edit file named on current line Alt+= : Execute the current line or marked set of lines as commands | | |
| -----Ctrl Keys----- | | |
| Ctrl : change function line descriptions Ctrl+Backspace : delete a line Ctrl+Enter : like Enter except no new line | | |
| Page Down | Page Up | F3=Exit |

| E Editor Help | Ctrl+ Keys (Ctrl+A - Ctrl+Z) | Page 5 |
|---|------------------------------|---------|
| Ctrl+A : change window style Ctrl+D : DELETE word Ctrl+E : ERASE to end of line Ctrl+F : repeat previous FIND command Ctrl+H : split screen HORIZONTALLY Ctrl+L : copy text LINE to command line Ctrl+M : toggle tiled (non-overlapping)/MESSY (overlapping) window style Ctrl+Q : Swap to/from .ALL file (see ALL command) Ctrl+R : RECORD key sequence Ctrl+T : play recorded key sequence Ctrl+V : split screen VERTICALLY Ctrl+W : move to next WINDOW Ctrl+X : force syntax expansion Ctrl+Z : ZOOM window to full screen | | |
| Page Down | Page Up | F3=Exit |

File EHELP.HLP (3/5)

| | | |
|--|--------------------------|---------|
| E Editor Help | Command Summary (1 of 4) | Page 6 |
| <p>Command Syntax</p> <hr style="width: 50%; margin-left: 0;"/> <pre> [L] /pattern/[c e] [m a] [- +] [r f] - locates string C /old/new/[c e] [m a] [- +] [r f] - changes string c = ignore case e = match case exactly m = within marked area a = all file - = backwards in file + = forward in file r = search Right to Left f = search from Left to Right NAME, N [filespec] - renames for next save PRINT - prints current file SAVE, S [filespec] - saves file PUT filespec - saves marked area to file APPEND [filespec] - appends marked area to file GET filespec - gets and inserts file </pre> | | |
| Page Down Page Up | | F3=Exit |

| | | |
|---|--------------------------|---------|
| E Editor Help | Command Summary (2 of 4) | Page 7 |
| <p>Command Syntax</p> <hr style="width: 50%; margin-left: 0;"/> <pre> BOX [1 2 C P A E R] KEY ##### [character] - repeats key horizontally LOOPKEY ##### ALL [character] - repeats key vertically LIST, FINDFILE filespec - loads a list of files MATCHTAB [ON OFF] EXPAND [ON OFF] MArgins # # # - left, right margin, new-paragraph TABS [n1 n2 n3 ... n20] - sets tab at column listed AUTOSAVE [#####] - autosaves file after ##### lines WS 1 2 3 4 - sets window style ALL /pattern[/[e]] - creates a new file called .ALL showing all occurrences of the pattern given </pre> | | |
| Page Down Page Up | | F3=Exit |

| | | |
|--|--------------------------|---------|
| E Editor Help | Command Summary (3 of 4) | Page 8 |
| <p>Command Syntax</p> <hr style="width: 50%; margin-left: 0;"/> <pre> DRAW [1] [2] [3] [4] [5] [6] [B] [/any character] MATH expression Operators -, +, *, /, % Numbers decimal, hex (leading x), octal (leading o) Example: math 10*30/5 MATHX - same as math, result shown in hex MATHO - same as math, result shown in octal ASC [char] - gives ASCII value of character (e.g. ASC X=88). If no argument specified, uses current text character CHR ### - shows character corresponding to ASCII # ADD - adds marked column of numbers </pre> | | |
| Page Down Page Up | | F3=Exit |

File EHELP.HLP (4/5)

| | | |
|---|--------------------------|---------|
| E Editor Help | Command Summary (4 of 4) | Page 9 |
| <u>Command Syntax</u> | | |
| SORT - sorts marked area or all | | |
| SIZE - changes window size | | |
| DRAG - moves window | | |
| #### - goes to line #### | | |
| +[#] - down # lines (or End of file) | | |
| -[#] - up # lines (or Top of file) | | |
| AUTOSHELL [ON OFF] | | |
| [DOS] [command] - any DOS command can be typed on the command line if AUTOSHELL is on | | |
| Page Down | Page Up | F3=Exit |

| | | |
|--|---------------|---------|
| E Editor Help | Edit Commands | Page 10 |
| <u>How To Edit Additional Files during one EDIT session:</u> | | |
| EDIT,ED,E [d:] [\path\]filename.ext | | |
| Loads the file if it is not already loaded. | | |
| Typing the EDIT command without the file specification switches you to the next loaded file in the ring (works the same as pressing F10.) After the last file, you are returned to the first file in the ring of loaded files. | | |
| Page Down | Page Up | F3=Exit |

| | | |
|--|-----------------------------|---------|
| E Editor Help | Editor Termination Commands | Page 11 |
| <u>How to get out of the E Editor</u> | | |
| FILE, F command (F4) saves and then removes file from memory. | | |
| QUIT, Q command (F3) removes file from memory without saving. | | |
| If QUIT is entered after file has been altered, the message: | | |
| "Quit without saving? Press Y or N" | | |
| is displayed on the bottom line. Typing N cancels the QUIT, typing Y discards all changes since the last SAVE or FILE. | | |
| EXIT quits all files without saving and exits the E Editor. Use with care! | | |
| Page Down | Page Up | F3=Exit |

File EHELP.HLP (5/5)

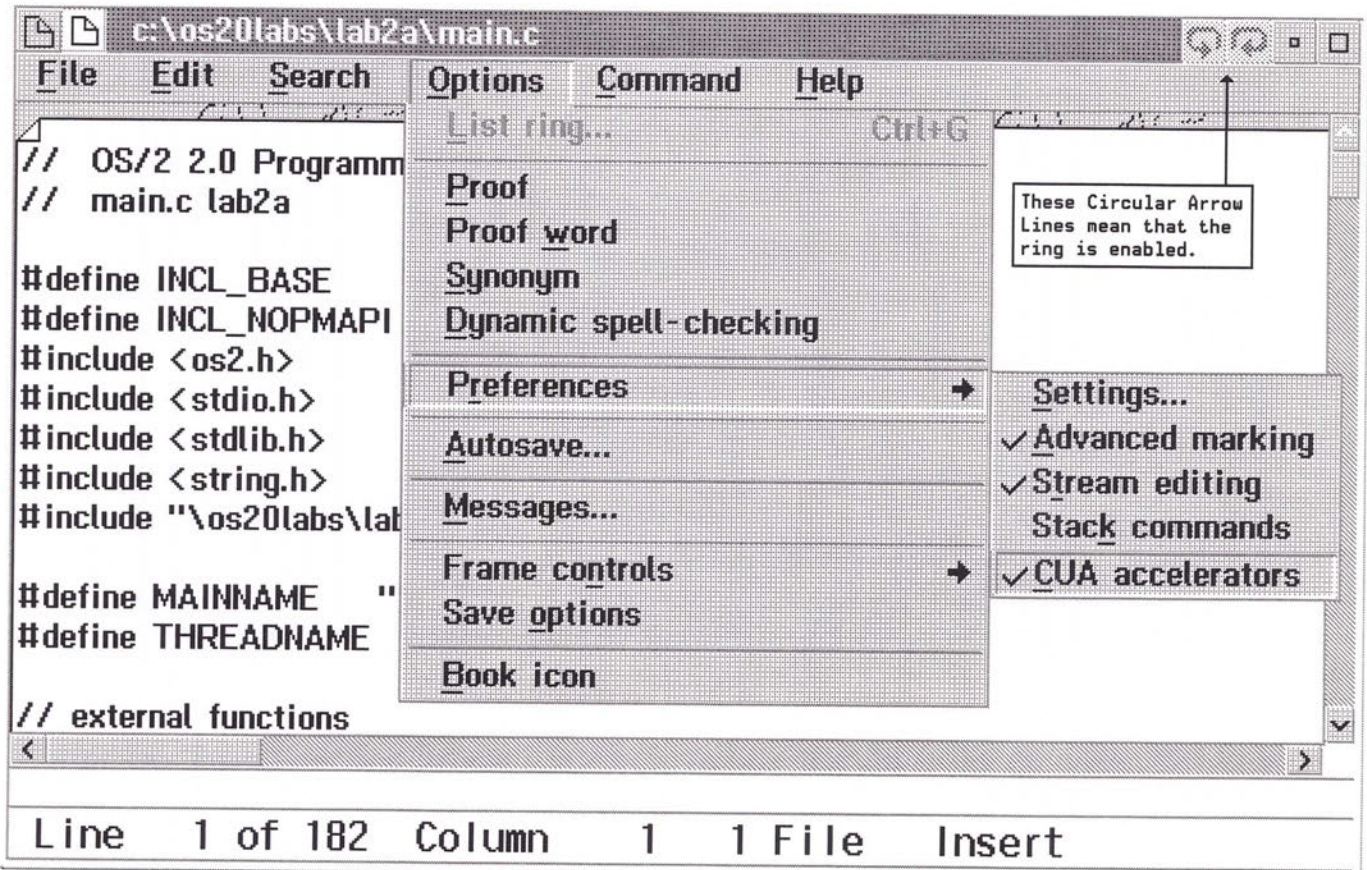
| E Editor Help | Cursor Movement | Page 12 |
|--------------------|--|---------|
| Up, Down Arrows | Moves one line up/down. | |
| Left, Right Arrows | Moves one character to the left/right. | |
| Home | Moves to column 1 of the current line. | |
| End | Moves to the end of the current line. | |
| Page Up | Displays text above current page. | |
| Page Down | Displays text below current page. | |
| Ctrl+Home | Moves to top line of file. | |
| Ctrl+End | Moves to bottom line of file. | |
| Tab, Shift+Tab | Moves to next and previous tab stops. | |
| Ctrl+PgUp | Moves to top of screen. | |
| Ctrl+PgDn | Moves to bottom of screen. | |
| Ctrl+Left Arrow | Moves to beginning of word left of cursor. | |
| Ctrl+Right Arrow | Moves to beginning of word to right of cursor. | |
| Ctrl+Enter | Moves to column 1 of next line. | |
| Page Down | Page Up | F3=Exit |

| E Editor Help | Copying, Moving, Deleting Text | Page 13 | | | | | | | | | | | | |
|--|--------------------------------------|---------|---------------|------------------------|--|------------------|-----------------------------|------------------|------------------------------|--------------------|------------------------|---------------------|------------------|--------------------------------------|
| <ol style="list-style-type: none">1. Press a Mark key (see list on left) once at the beginning of the text and again at the end of the text to highlight the text to be copied, moved, or deleted.2. To copy or move, move the cursor to the desired destination.3. Press the operator keys (see list on right). | | | | | | | | | | | | | | |
| <table><thead><tr><th>TEXT MARKERS:</th><th>MARKED TEXT OPERATORS:</th></tr></thead><tbody><tr><td>Alt+L: LINE mark for one line or paragraph</td><td>Alt+C: COPY mark</td></tr><tr><td>Alt+Z: mark, character mode</td><td>Alt+M: MOVE mark</td></tr><tr><td>Alt+B: BLOCK mark rectangles</td><td>Alt+D: DELETE mark</td></tr><tr><td>Alt+U: UNMARK any area</td><td>Alt+O: OVERLAY mark</td></tr><tr><td>Alt+W: WORD mark</td><td>Alt+A: ADJUST mark (Alt+O with fill)</td></tr></tbody></table> | | | TEXT MARKERS: | MARKED TEXT OPERATORS: | Alt+L: LINE mark for one line or paragraph | Alt+C: COPY mark | Alt+Z: mark, character mode | Alt+M: MOVE mark | Alt+B: BLOCK mark rectangles | Alt+D: DELETE mark | Alt+U: UNMARK any area | Alt+O: OVERLAY mark | Alt+W: WORD mark | Alt+A: ADJUST mark (Alt+O with fill) |
| TEXT MARKERS: | MARKED TEXT OPERATORS: | | | | | | | | | | | | | |
| Alt+L: LINE mark for one line or paragraph | Alt+C: COPY mark | | | | | | | | | | | | | |
| Alt+Z: mark, character mode | Alt+M: MOVE mark | | | | | | | | | | | | | |
| Alt+B: BLOCK mark rectangles | Alt+D: DELETE mark | | | | | | | | | | | | | |
| Alt+U: UNMARK any area | Alt+O: OVERLAY mark | | | | | | | | | | | | | |
| Alt+W: WORD mark | Alt+A: ADJUST mark (Alt+O with fill) | | | | | | | | | | | | | |
| Page Up | | F3=Exit | | | | | | | | | | | | |

The Enhanced Editor

The on line instructions for EPM are available by entering **VIEW EPMUSERS** as shown on page EDIT-1. File EPMUSERS.INF is available on the Developer's Connection CD ROM. The version of EPM shown here in the screen snapshots is the 6.0 BETA from Volume IV of the Developer's Connection.

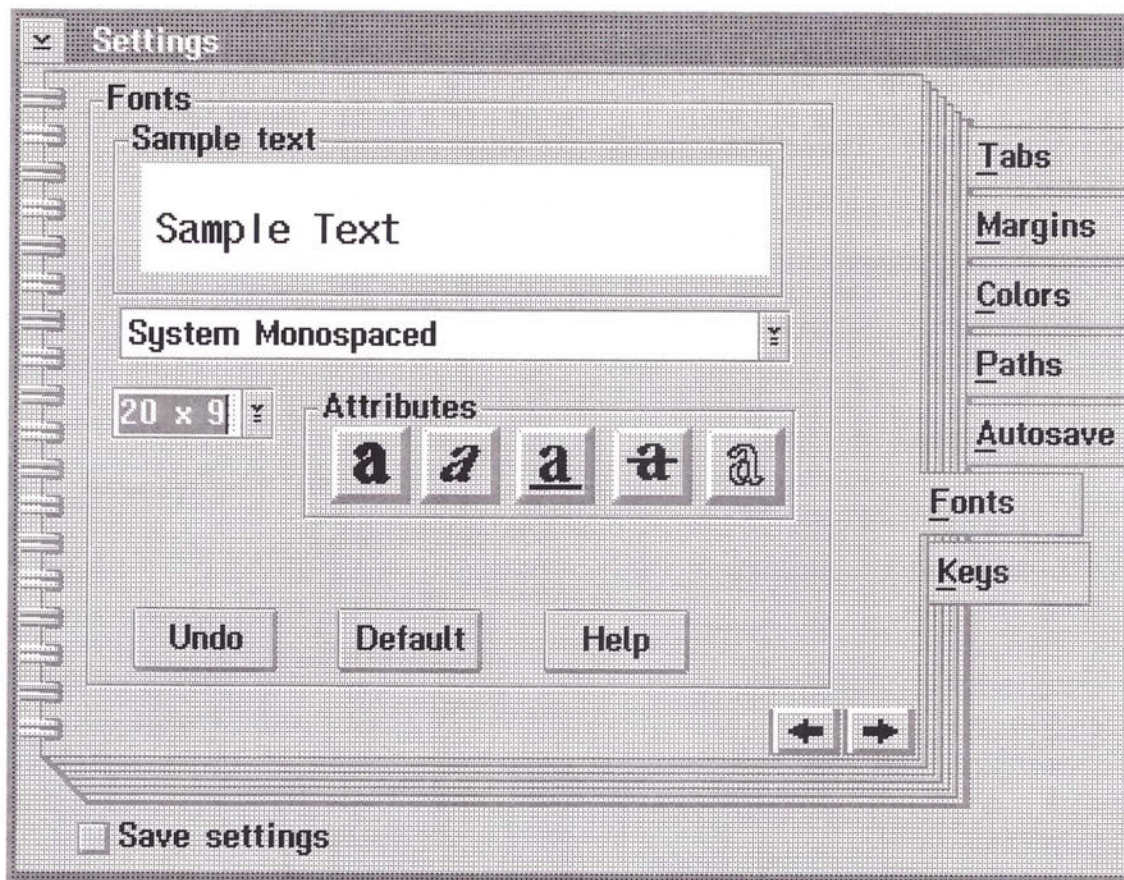
EPM can be configured to accept the same line marking and block marking commands that E.EXE (and PE2, etc) accept. We will set it up that way. EPM allows you to select a font style and size. For editing programs I prefer to use a fixed spaced font. Let's look at the setup of EPM. These illustrations may differ slightly from the version of EPM that you are using.



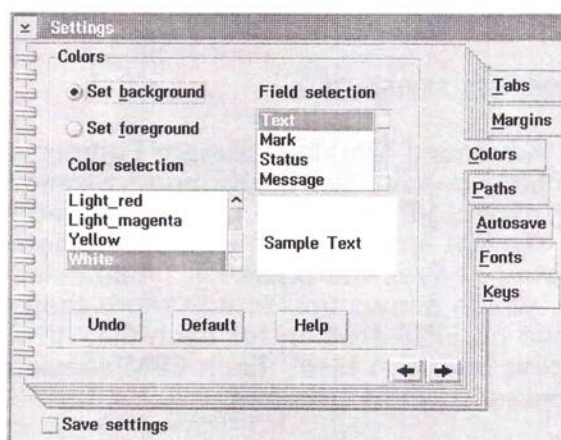
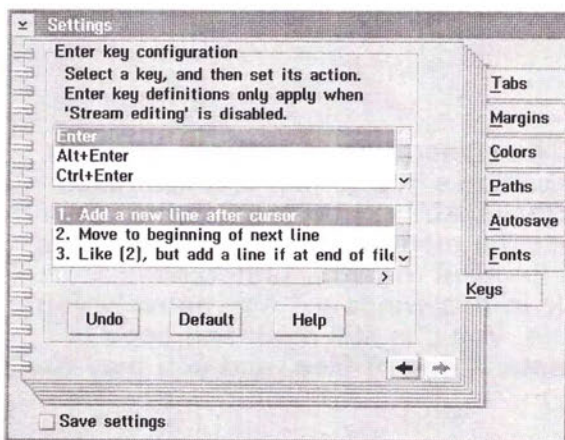
SETTING PREFERENCES

Select Advanced Marking, Stream Editing, and CUA accelerators. Advanced Marking makes EPM behave like E.EXE and. Stream Editing allows you to join and split lines in the "expected" manner. CUA accelerators will make ALT + C go to "Command" in the menu. To get a description of these options, select the menu item (CUA accelerators is selected above) and press F1. Also select **Ring Enabled** (on *your* preferences menu), which allows you to edit more than one file in this window. Alternatively you can have on EPM session for each file. In that case, you can still mark and copy or move text between files. **Each EPM session can have a ring of files, and you may run any number of EPM sessions.**

The Settings dialog allows you to set Fonts, Colors, Keys, etc. Use the **Fonts** setting to select a system monospaced font for editing programs. That way the comments line up. Select a font which is large enough to be easy to read.

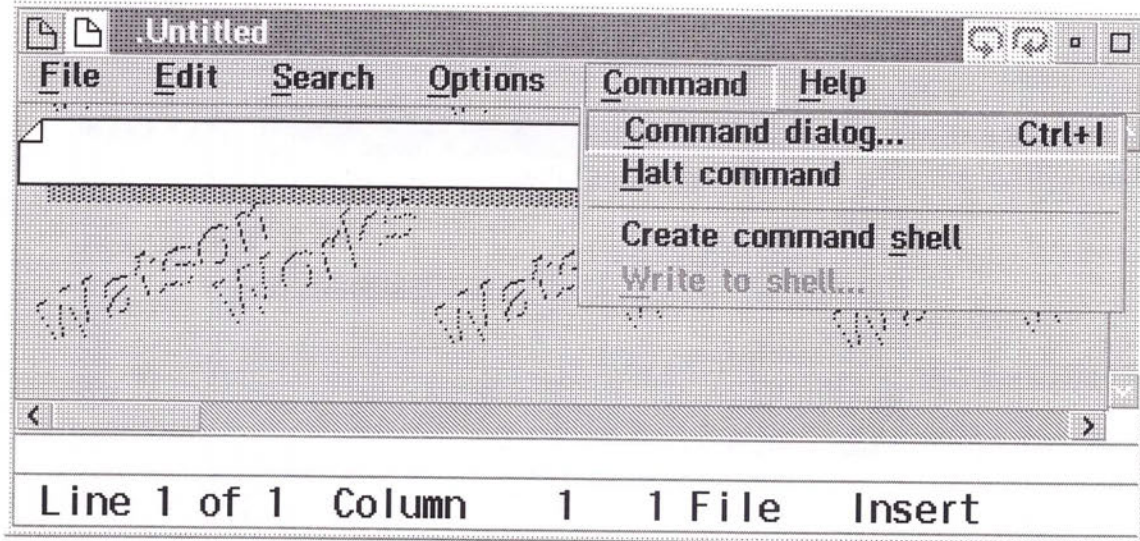


Use the Keys setting to set the function of the ENTER key(s), and the Colors setting to set a pleasant foreground and background color.

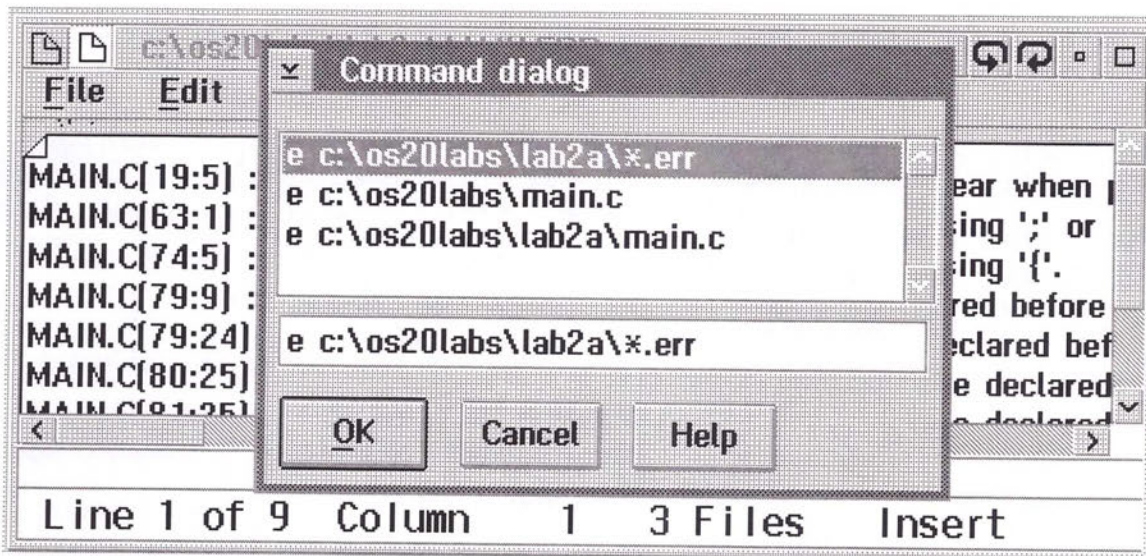


Using the Command [line] Dialog

There are some of us who still like to type commands into our otherwise mouse driven editor. The "standardized" commands from the E editor still work with EPM. You can bring up the command dialog by pulling down **Command** and selecting **Command dialog**, or you can simply enter CTRL+I.

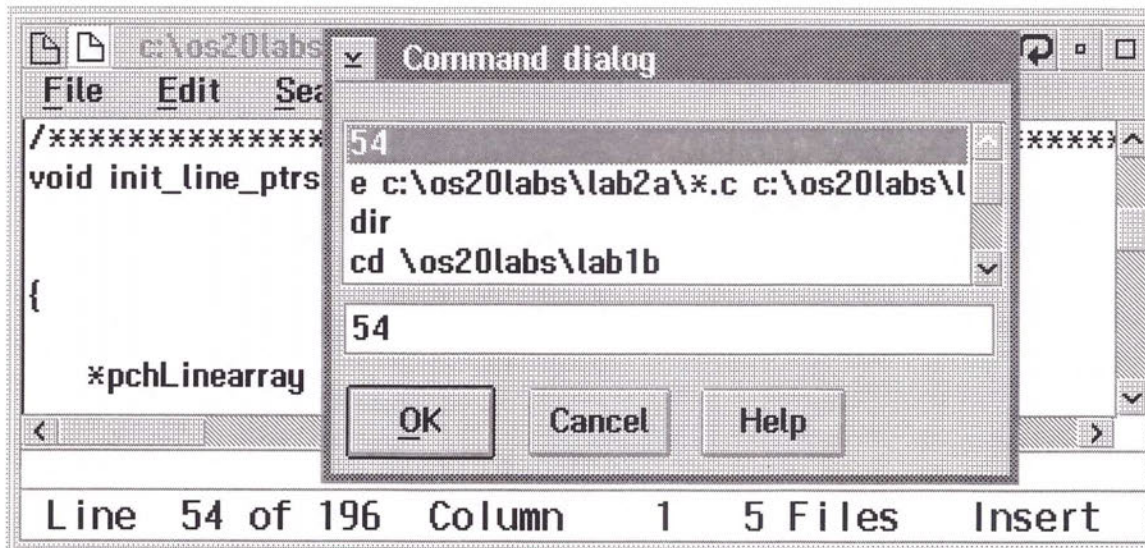


The command dialog lets you enter commands, and remembers the commands you have previously entered so you can select them without re-entering them. You can use the command dialog to quickly go to a line number, to add a set of files to the ring, etc.



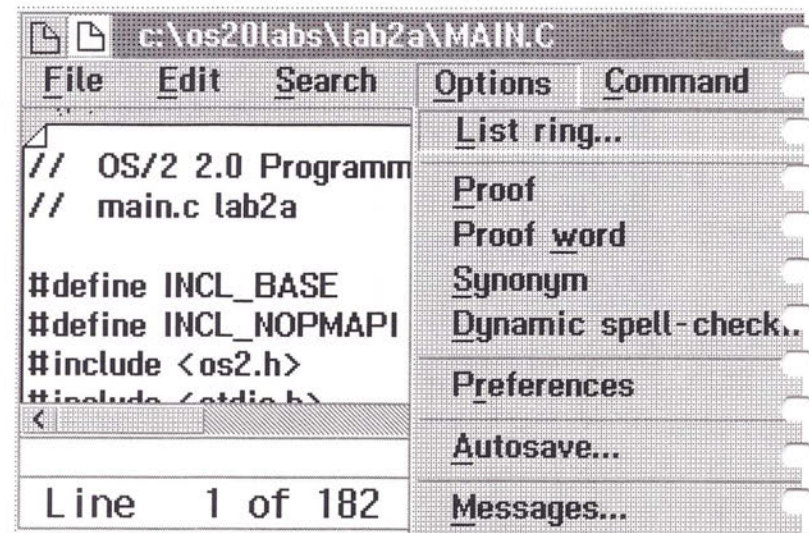
COMMAND DIALOG - GO TO A LINE NUMBER

The example below shows you how to go to a specific line number. You can also enter OS2 commands by typing OS2 on the command line and hitting enter or selecting OK.



OPTIONS - LIST RING

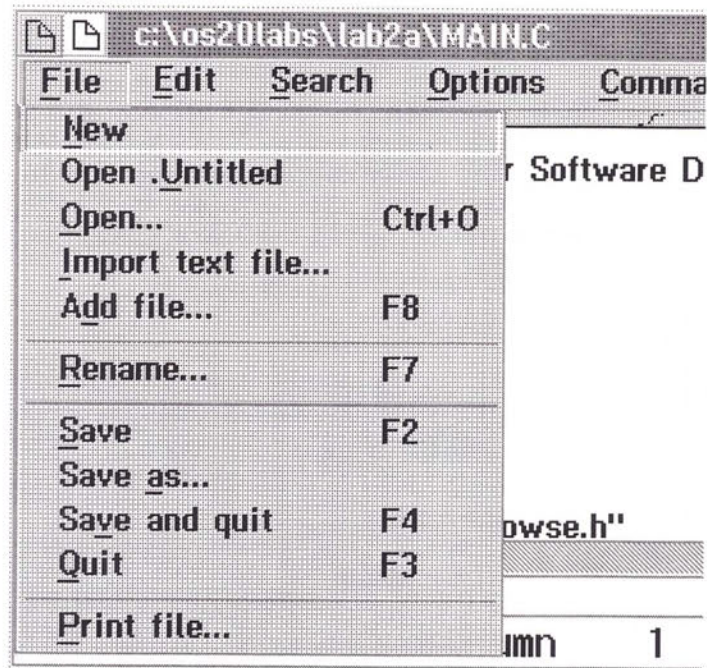
You can select **Options** and **List Ring** (or just enter CTRL + G) and see the names of the files in the ring and select a file. You can also click on the ring icons or use F11 or F12 to go forward or backwards around the ring. Remember that you can have any number of EPM sessions up and each session may have its own ring. You can quit files with F3 or save in quit with F4, eliminating them from the ring. See the **File** Pull-down.



FILE MENU - ACCELERATORS, OPEN vs ADD

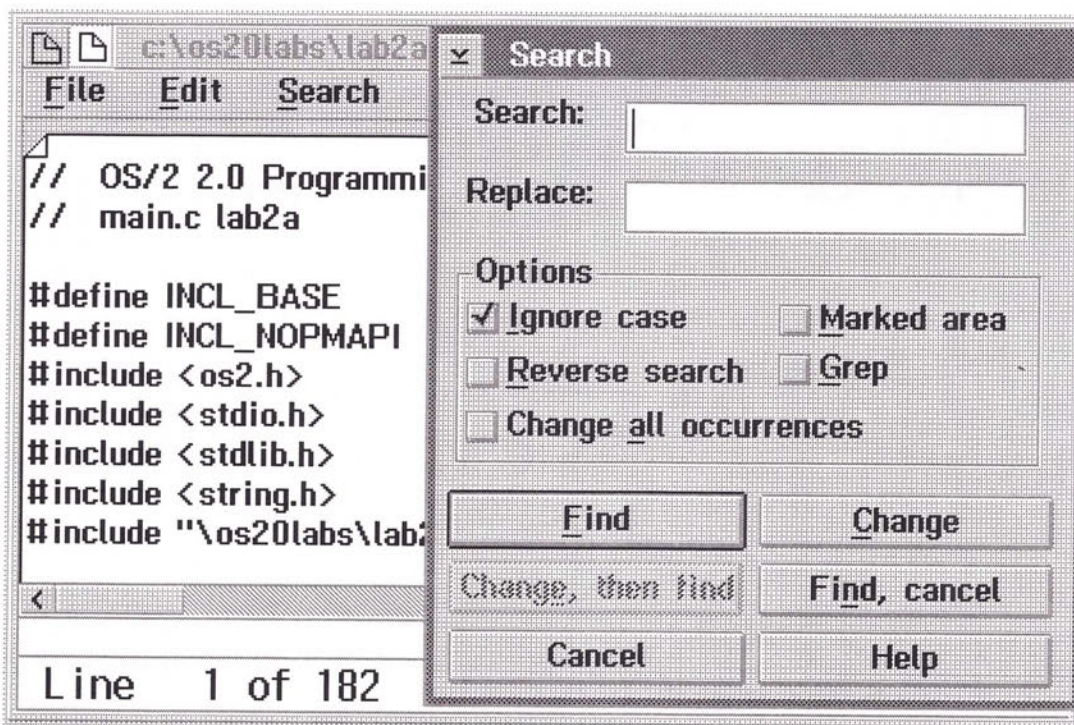
The file menu lists function keys which operate regardless of whether or not the menu is visible. Technically these are called accelerators. **Add file** lets you add another file to the ring of the current EPM session, while the **Open** starts a new EPM session. The function keys are similar to those for the E editor as shown on page EDIT-3.

You can get help on these accelerator keys by going to **Help** and then **Keys**. A list of these appears at the end of the chapter.



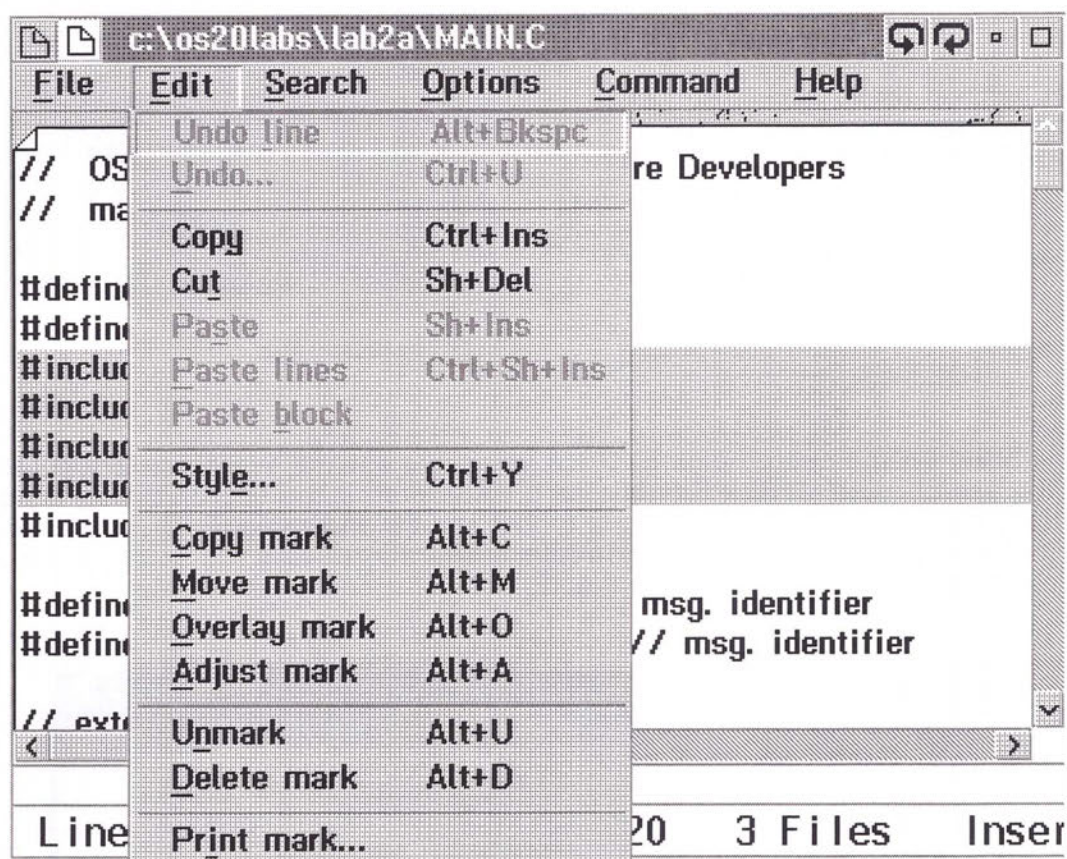
SEARCH

CTRL+S will bring up the search dialog. Once you have set the search criterion, you can repeat the search with CTRL+F (same as E.EXE) even if the search dialog is not visible. You can also use this dialog to do a global search and replace. Or, if you prefer, you can do the search and the search-and-replace operation using the command dialog with the same commands you used with E.EXE.



LINE AND BLOCK MARKING, COPY, MOVE, SHIFT, DELETE, CUT, PASTE

With advanced marking enabled, you can mark lines using ALT+L (use another ALT+L to mark a group of lines). You can delete the marked lines, copy them, or move them using the same keys as you would with the E editor. You can do these operations on a single file, between files in the ring, and between EPM sessions. ALT+Y will move the cursor to the mark. Line, Block and Character marking is supported, as it is in E.EXE.



Selected Key Definitions

| | |
|----------|--|
| F1 | Displays help for selected item |
| F2 | Saves the current file. |
| F3 | Quits the current file. |
| F4 | Save and Quit the current file. |
| F5 | Starts a new EPM Session with a new file |
| F7 | Renames the current file. |
| F8 | Adds a file to the ring of the current EPM session. |
| F9 | Undo changes to current line |
| F10 | Switches between the menu bar and the file being edited. |
| F11 | Switches to the previous file in the edit ring. |
| F12 | Switches to the next file in the edit ring. |
| Alt+F1 | Inserts a list of various graphic characters at the current cursor position. |
| Alt+A | Move Mark Itself to current cursor position. |
| Alt+B | Block-marks the current cursor position. |
| Alt+C | (*) Copies the marked text. |
| Alt+D | Deletes the currently marked characters. |
| Alt+E | (*) Moves the cursor to the end of the current mark. |
| Alt+F | (*) Fills marked area |
| Alt+J | Joins the next line with line at the current cursor position |
| Alt+L | Line-marks the line at the current cursor position. |
| Alt+M | Moves the marked text. |
| Alt+N | Types the current file name at the current cursor position. |
| Alt+O | (*) Overlay the current mark. |
| Alt+P | Reformats the paragraph from current cursor position, using current margins. |
| Alt+R | Reformats a marked block of text to a new set of margins. |
| Alt+S | (*) Splits the line at the current cursor position. |
| Alt+T | Centers the text within the current mark (or margins, if no mark). |
| Alt+U | Unmark. |
| Alt+W | Marks the word at the current cursor position using a character mark. |
| Alt+Y | Moves the cursor to the beginning of the current mark. |
| Alt+Z | Character-marks the current character (use two of these). |
| Alt+1 | Edits the file name on the line at the current cursor position. |
| Ctrl+F1 | Converts current word to uppercase. |
| Ctrl+F2 | Converts current word to lowercase. |
| Ctrl+F3 | Converts marked text to uppercase. |
| Ctrl+F4 | Converts marked text to lowercase. |
| Ctrl+F5 | Moves the cursor to the beginning of the current word. |
| Ctrl+F6 | Moves the cursor to the end of the current word. |
| Ctrl+F7 | Moves marked text to the left one column. (Block or line marks only.) |
| Ctrl+F8 | Moves marked text to the right one column. (Block or line marks only.) |
| Ctrl+C | Repeats the last CHANGE command. |
| Ctrl+D | Erases from the current cursor position to the beginning of the next word. |
| Ctrl+E | Erases from the current cursor position to the end of the line. |
| Ctrl+F | Repeats the last LOCATE command. |
| Ctrl+I | Brings up the EPM command dialog. |
| Ctrl+L | Copies the line at the current cursor position to the command line. |
| Ctrl+S | Activates the Search dialog. |
| Ctrl+BKS | Deletes the current line. |
| Del | Deletes the character at the current cursor position. |
| Ins | Switches between insert and replace modes. |

(*) These definitions will be blocked if the CUA Accelerators option is selected. Instead, the key will activate the corresponding action bar menu item.

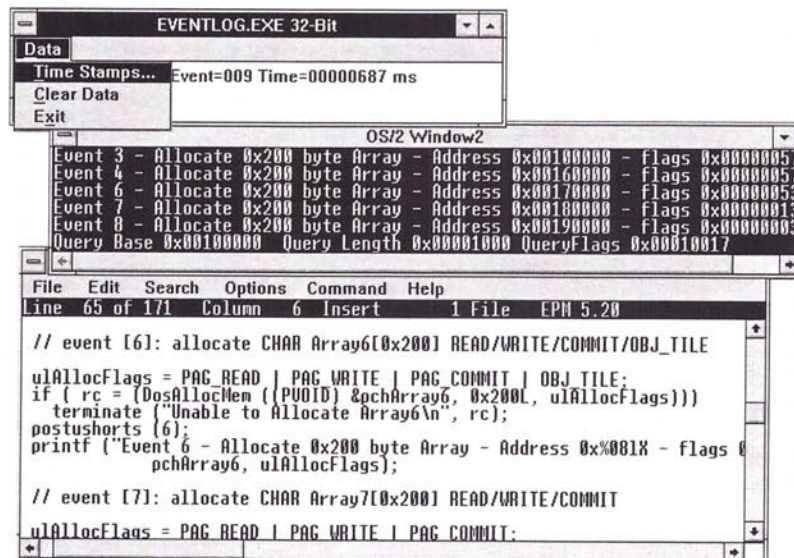
Selected Commands

| | |
|-----------|--|
| [#] | Go To Line |
| ASC | Get ASCII Value |
| BOTTOM | Go To Bottom |
| CD | Change Directory |
| CENTER | Center Mark |
| CHANGE | Change Text |
| CHR | Get Character for ASCII Value |
| CLOSE | Close the Edit Window |
| COPY2CLIP | Copy to Clipboard |
| DIR | Directory List |
| EDIT | Edit File - Add file to ring |
| FILE | Save and Quit |
| FILL | Fill Mark |
| GET | Add named file text to current file |
| LOWERCASE | Convert Marked Text to Lowercase |
| MARGINS | Set Text Margins |
| MATCHTAB | Use Words as Tab Stops |
| MATHx | Calculate |
| MONOFONT | Change to monospaced font |
| NAME | Rename Current File |
| NEWWINDOW | Move Current File to a New Edit Window |
| OPATH | Open a File in a PATH Setting |
| OPEN | Open New Edit Window |
| OS2 | Process OS/2 Command |
| PASTE | Paste Text |
| PATH | Show PATH Setting |
| PRINT | Print File |
| QUIT | Quit File |
| QD | Query Date |
| QT | Query Time |
| SAVE | Save File |
| SET | Show Environment Settings |
| SHELL | Start a Command Shell |
| SORT | Sort Marked Lines |
| TABS | Set Tab Stops |
| TOP | Go To Top |
| UPPERCASE | Convert Marked Text to Uppercase |
| VER | Show Editor Version |
| VOL | Show Volume Label |

OS/2 2.0 for Application Developers

Chapter L1

Laboratory Project One



The image shows two overlapping windows from an OS/2 environment. The top window is titled "EVENTLOG.EXE 32-Bit" and displays event data for "Event=009 Time=00000687 ms". The bottom window is titled "OS/2 Window2" and shows a source code editor with the following code:

```
File Edit Search Options Command Help
Line 65 of 171 Column 6 Insert 1 File EPH 5.20

// event [6]: allocate CHAR Array6[0x200] READ/WRITE/COMMIT/OBJ_TILE
ulAllocFlags = PAG_READ | PAG_WRITE | PAG_COMMIT | OBJ_TILE;
if ( rc = (DosAllocMem ((PVOID) &pchArray6, 0x200L, ulAllocFlags)))
    terminate ("Unable to Allocate Array6\n", rc);
postushorts (6);
printf ("Event 6 - Allocate 0x200 byte Array - Address 0x%08lX - flags 0
        pchArray6, ulAllocFlags);

// event [7]: allocate CHAR Array7[0x200] READ/WRITE/COMMIT
ulAllocFlags = PAG_READ | PAG_WRITE | PAG_COMMIT;
```

By Charles R. Chernack

Contents

| | |
|--|-------|
| Laboratory Project One | L1-3 |
| Part A - 16 Bit Application | L1-4 |
| Analysis of PM Application EVENTLOG.C | L1-5 |
| Analysis of OS/2 Application LAB1A.C | L1-7 |
| Part B - Conversion 16-Bit to 32-Bit Application | L1-9 |
| Warm-Up Questions | L1-10 |
| Programming Steps | L1-11 |
| Follow-Up Questions | L1-12 |
| Part C - Allocation of Private Memory Objects | L1-14 |
| Warm-Up Questions | L1-15 |
| Programming Steps | L1-16 |
| Follow-Up Questions | L1-17 |
| Part D - Allocation of Shared Memory Objects | L1-19 |
| Warm-Up Questions | L1-20 |
| Programming Steps | L1-21 |
| Follow-Up Questions | L1-23 |
| Part E - Accessing Shared Memory Objects | L1-26 |
| Organization of Part E | L1-27 |
| Programming and Analysis of Part E | L1-29 |
| Part F - SubAllocation | L1-33 |
| Organization of Part F | L1-34 |
| Programming and Analysis of Part F | L1-36 |
| Answers to Questions | L1-38 |
| Lab Hints | L1-56 |

NOTES:

This is the table of contents for lab project one from the "advanced" OS/2 2.0 programming class. Now that you have taken this class, you are well able to deal with these labs. The labs are very tutorial and as you can see each lab is about 50 pages. There is no dependency between labs, so you can do only those parts which are of interest to you. Part A is a 16-bit OS/2 and a 16-bit Presentation Manager application which talk to each other. In Part A you just do an analysis of the code. Even if you do not know Presentation Manager, you can review the answers to the questions!

Part B involves converting the 16-bit code to 32-bit code. Since named shared memory and WinPostQueueMsg are compatible between 16 and 32-bit applications, you can convert either or both of the 16-bit applications and they will continue to work.

Parts C, D, E and F are memory allocation exercises. Part F covers inter-process suballocation and freeing of memory, and is a good exercise. To print the text of these labs, unzip disk#4 (see page 4 at the beginning of the supplement) and copy the files using the /B option to a laser printer supporting Hewlett Packard PCL5. The Laserjet 3 or 4 may be used.

OS/2 2.0 for Application Developers

Chapter L2

Laboratory Project Two



by Charles R. Chernack

Contents

| | |
|--|-------|
| Laboratory Project Two | L2-1 |
| Overview | L2-3 |
| Part A - 32-Bit Semaphores, Threads and Timers | L2-4 |
| Objectives and Block Diagram | L2-4 |
| Warm-Up Questions | L2-5 |
| Programming Steps | L2-7 |
| Part B - Thread Priority | L2-10 |
| Objectives, Overview, API Calls | L2-10 |
| Block Diagram | L2-11 |
| Warm-Up Questions | L2-12 |
| Programming Steps | L2-15 |
| Part C - Thread Synchronization | L2-18 |
| Objectives, Overview, API Calls | L2-18 |
| Source Files and Structures | L2-19 |
| Programming Steps | L2-21 |
| Review Questions | L2-27 |
| Part D - MuxWait Semaphore Setup | L2-30 |
| Overview | L2-30 |
| API Calls | L2-31 |
| Warm-Up Questions | L2-33 |
| Programming Steps | L2-37 |
| Part E - Edge/Level Operation of Event Semaphore | L2-44 |
| Objectives, Overview, API Calls | L2-44 |
| Programming Steps | L2-45 |
| Answers to Questions | L2-47 |
| Lab Hints | L2-67 |

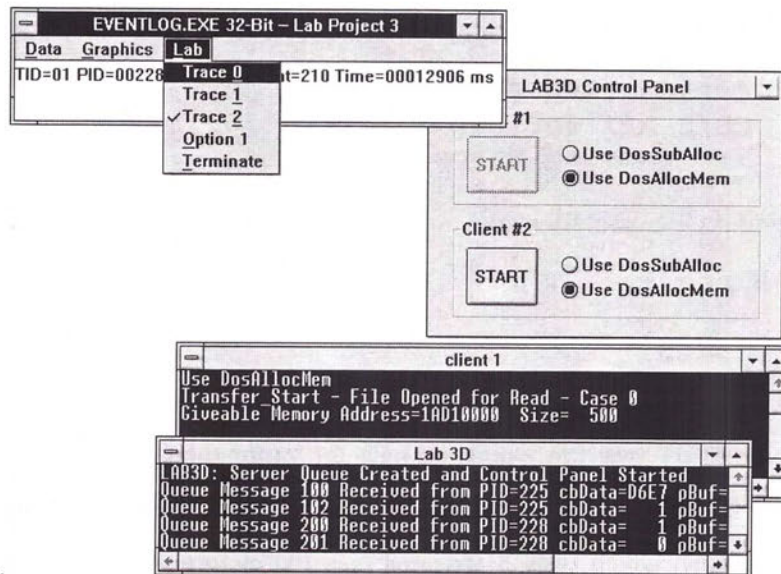
NOTES:

The best parts of this lab are parts C, D and E. But the entire lab is a good programming exercise and develops the code for the "walk thread" demonstration programs. If you are short on time just do part C or parts C and D.

OS/2 2.0 for Application Developers

Chapter L3

Laboratory Project Three



by Charles R. Chernack

Contents

| | |
|--|-------|
| Laboratory Project Three | L3-1 |
| Overview | L3-3 |
| Part A - malloc, _alloca, DosSubAllocMem | L3-5 |
| Overview | L3-5 |
| OS/2 API for Part A | L3-5 |
| C Functions for Part A | L3-6 |
| Technical Note on malloc, _alloca | L3-6 |
| Technical Note on ANSI Sequences | L3-7 |
| Line/Grid Plotting in EVENTLOG/TOOLS.C | L3-8 |
| Terminate/Trace Options in EVENTLOG/STRUCS.H | L3-9 |
| Warm-Up Questions | L3-10 |
| Programming Steps | L3-13 |
| Review Questions | L3-17 |
| Part B - Conversion of Lab 2 Part D to Multithread | L3-19 |
| Overview | L3-19 |
| C Functions for Part B | L3-19 |
| Technical Note on C Set/2 Libraries | L3-20 |
| Programming Steps | L3-23 |
| Part C - Buffering of Stdout | L3-27 |
| Overview | L3-27 |
| C Functions for Part C | L3-27 |
| Technical Note on C Set/2 File Handling | L3-28 |
| Programming Steps | L3-29 |
| Part D - Data Transport through Queues | L3-32 |
| Overview | L3-32 |
| C Functions for Part D | L3-33 |
| OS/2 API for Lab 3 Part D | L3-34 |
| Programming Steps | L3-35 |
| Answers to Questions | L3-50 |
| Lab Hints | L3-66 |

NOTES:

Part A generates performance graphs on malloc, _alloca, and DosSubAllocMem. It is an interesting programming exercise. If you want to know exactly how long something takes, you may use the timing features of IPMD or ASDT (see the workshop help for information on ASDT).

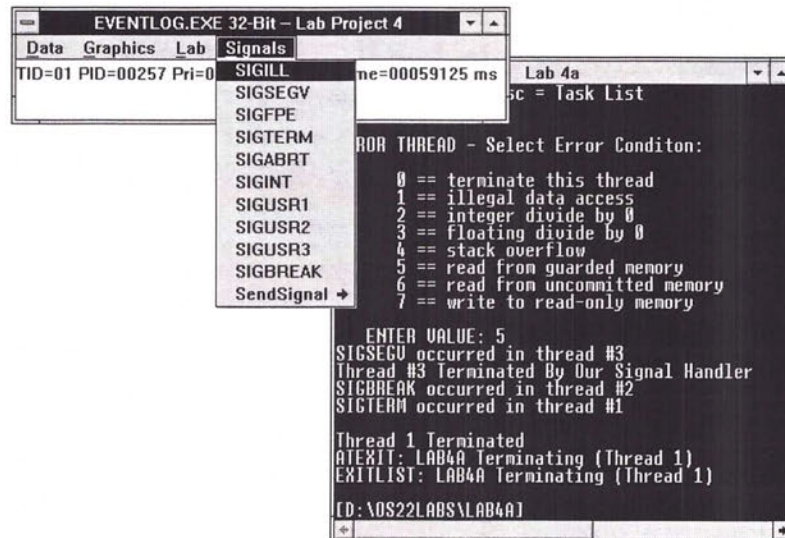
Skip part B. Part C is a directed demonstration of the buffering of standard out. I'd skip it.

Part D is an excellent lab which uses 5 sessions (see the picture on the previous page), and which deals with the relationship between the sessions. All the code is written for you except for parts of LAB3D.C. LAB3D.EXE is a server program using an OS/2 queue. The control panel and eventlog are presentation manager applications. The server communicates with the control panel and eventlog using named shared memory and also using WinPostQueueMsg (to put messages into their PM message queue). When one of the "start" pushbuttons is depressed, a queue message is sent from the control panel to the server which in turn starts one of the clients. The clients transfer a file to the server (say copying BSEDOS.H) and then terminate. When the client terminates, the server sends a message to the control panel to reactivate the "start" pushbutton. A great lab!

OS/2 2.1 for OS/2 Developers

Chapter L4

Laboratory Project Four



by Charles R. Chernack

Contents

| | |
|--|-------|
| Laboratory Project Four | L4-1 |
| Overview | L4-3 |
| Part A - Signal Handling | L4-6 |
| Overview | L4-6 |
| C Functions for Part A | L4-7 |
| Technical Notes on Signals | L4-8 |
| Warm-Up Questions | L4-10 |
| Programming Steps | L4-12 |
| Part B - Signals for Error Recovery | L4-21 |
| Overview | L4-21 |
| Setting up for <code>_beginthread()</code> | L4-22 |
| Programming Steps | L4-23 |
| Part C - Building a Dynamically Linked DLL | L4-29 |
| Overview | L4-29 |
| IBM C Set/2 API for Part C | L4-29 |
| Technical Background | L4-30 |
| Programming Steps | L4-34 |
| Review Questions | L4-39 |
| Part D - Building a Statically Linked DLL | L4-41 |
| Overview | L4-41 |
| Programming Steps | L4-42 |
| Part E - Building a Subsystem | L4-43 |
| Overview | L4-43 |
| Warm-Up Questions | L4-44 |
| Programming Steps | L4-46 |
| Answers to Questions | L4-50 |
| Lab Hints | L4-66 |

NOTES:

Parts A and B demonstrate signals. In part A we raise signals, and in part B we cause exceptions which we deal with using signal handing. Part C is a great DLL lab which demonstrates how to export data pointers as well as procedure addresses. Part C also uses `#pragma data_seg` to create static shared-private and static shared_shared data. So I strongly recommend Part C if you want to learn more about DLLs.

Part D adds a few concepts which are interesting. I'd skip part E because the write-up is not complete. The solution to Part E is complete and it does illustrate a "subsystem" use of a DLL.

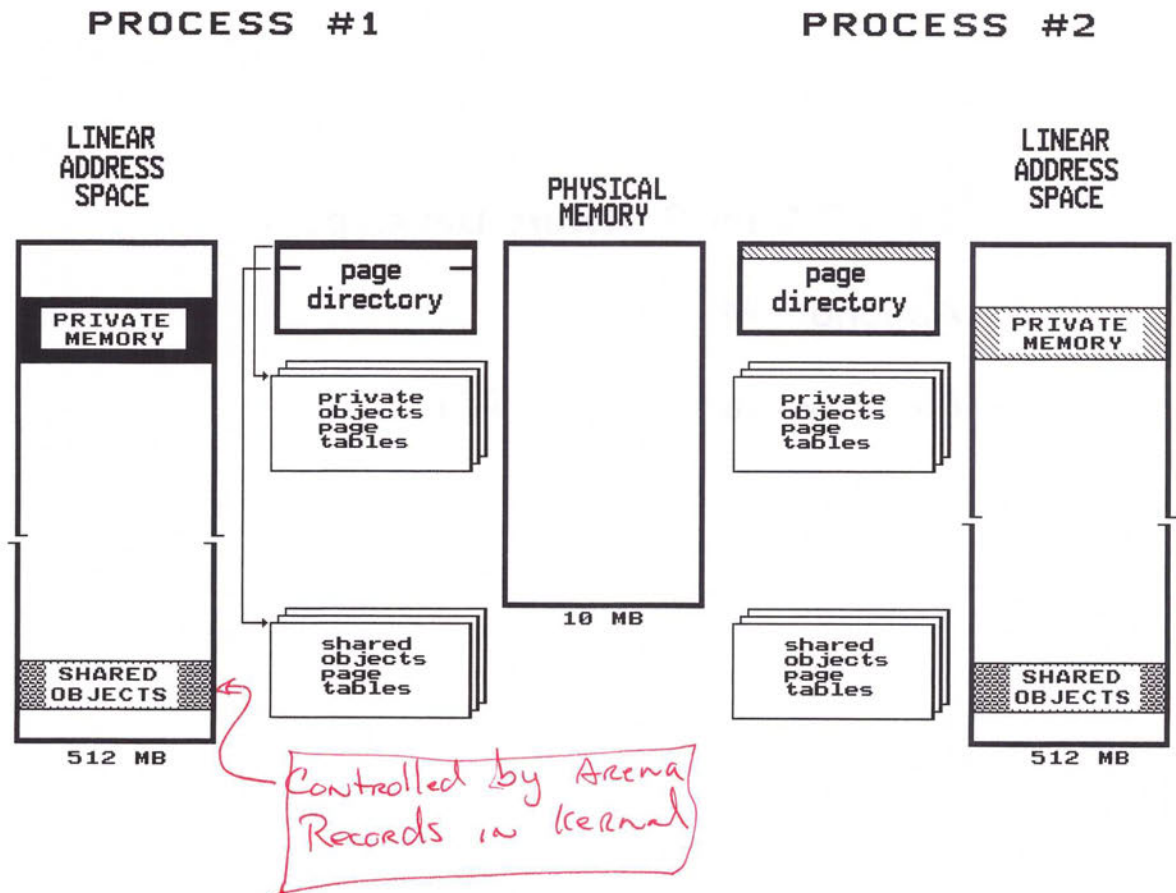
OS/2 2.1 for Software Developers

Appendix M

Memory Management API

by Charles R. Chernack

2 Process Address Space



NOTES:

Each process in 32-bit OS/2 may use up to 512 MB of virtual (linear) address space. The 512 MB limit comes from the restriction that all memory allocated by a 32-bit application, and all procedures called by a 32-bit application, must map into the 16-bit (tiled) address space.

Memory allocation in 32-bit OS/2 starts at 10000H (hex), or at the first 64K boundary. The first 64K of address space is not assigned due to a quirk of the CPU. Private objects -- that is objects which belong to this process and which are not shared with other processes -- are allocated from the low address space (shown at the top in the drawing). Shared objects are allocated from the high end of the address space towards the low end. This form of memory allocation is called "sparse allocation", in that the address space is not fully populated.

A design objective of 32-bit OS/2 was to move away from the Intel Protected Mode architecture. To this end, a FLAT CODE and DATA segments which start at 0 and are 512 MB long are established in the GDT. These GDT ordinals are the CS, DS, SS, and ES for all 32-bit processes.

3 Table of Contents

| | |
|--|------|
| Appendix M - Memory Management API | M-1 |
| Private Memory Allocation | M-4 |
| DosAllocMem | M-4 |
| DosQueryMem | M-6 |
| DosSetMem | M-7 |
| DosFreeMem | M-8 |
| Shared Memory Allocation | M-9 |
| Concepts of Shared Memory | M-9 |
| DosAllocSharedMem | M-10 |
| Gettable Shared Memory | M-11 |
| Giveable Shared Memory | M-12 |
| DosGetNamedSharedMem | M-13 |
| DosGiveSharedMem | M-14 |
| DosGetSharedMem | M-15 |
| Questions | M-16 |
| MEMDEM Demonstration Program | M-17 |
| Answers to Questions | M-18 |

DosAllocMem

```
DosAllocMem (PPVOID ppb, --> pointer to pointer
              ULONG cb,   <-- size in bytes
              ULONG flag); <-- allocation flags
```

rc =

| | |
|----|-------------------------|
| 0 | NO_ERROR |
| 8 | ERROR_NOT_ENOUGH_MEMORY |
| 87 | ERROR_INVALID_PARAMETER |
| 95 | ERROR_INTERRUPT |

flags from BSEMEMF.H

| | |
|-------------|-------------|
| PAG_READ | 0x00000001U |
| PAG_WRITE | 0x00000002U |
| PAG_EXECUTE | 0x00000004U |
| PAG_COMMIT | 0x00000010U |
| OBJ_TILE | 0x00000040U |

```
PCH pchArray6
```

```
ulAllocFlags = PAG_READ | PAG_WRITE | PAG_COMMIT | OBJ_TILE;
DosAllocMem ((PPVOID) &pchArray6, 0x200, ulAllocFlags);
```

NOTES:

DosAllocMem does not allocate physical memory. It allocates linear address space. This is "page table" space. If you specify PAG_COMMIT, then the first time you reference the memory actual RAM is allocated. This is called lazy commit.

The memory allocated is private to the process. Thus it starts in the low part of the linear address space. You cannot and do not specify the address of the memory. OS/2 2.1 finds free address space in the context of your process, and allocates that space. It then loads up your pointer to point to the beginning of your memory object.

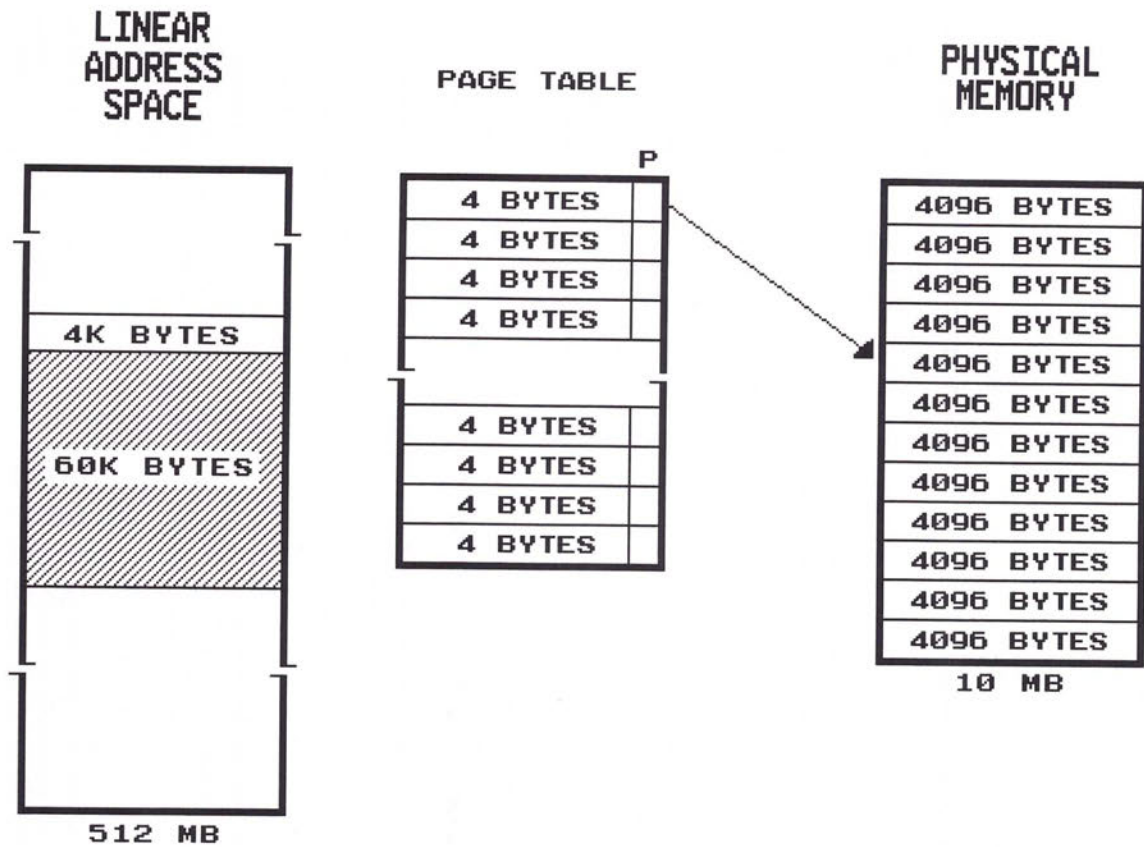
In OS/2 2.1 memory objects allocated with DosAllocMem are always object tiled, which means that the memory address starts on a 64K boundary. Whenever a memory object is allocated by 32-bit code, one or more entries are made in the 32-bit process's LDT. Thus 16-bit services may easily address and use the memory object. Even if you do not specify OBJ_TILE, DosAllocMem will consume a minimum of 64K of linear address space for each allocation. All memory objects are by default OBJ_TILE or **selector mapped**. That means that each object can map to the LDT and does start on a 64K boundary.

Prior to General Availability (GA) the PAG_COMMIT actually caused physical RAM or disk backing store to be allocated. This is no longer the case. PAG_COMMIT now makes the entire memory object ready for use: memory will be allocated as you use it.

The size of a memory object cannot be changed. A new object may be allocated just after this one on a subsequent call, and thus there is possibly no room for expansion. So you simply are not allowed to do it.

Every memory object is supported by an **arena record**. Each process has a **private arena**. There is one **shared arena** which contains one arena record per shared objects; and then there is a **system arena** for OS/2 system memory objects.

5 **Memory Allocation Quiz**



QUESTIONS:

- 1) If I allocate 200 bytes of read/write committed memory, how much memory can I access before I get a protection fault?
- 2) How much page table space is used by the 200 byte allocation?
- 3) If I allocate 2 64K arrays read/write committed, and they appear in consecutive 64K addresses, how much memory can I access starting at the beginning of the first array?
- 4) If I allocate 200 bytes of read/write committed memory, when will the present bit be set in the appropriate page table entry?

DosQueryMem

```

DosQueryMem (PVOID pb,          <-- starting query address
             PULONG pcb,       <-> size of region
             PULONG pFlag);    --> flags found

```

rc =

```

0      NO_ERROR
87     ERROR_INVALID_PARAMETER
95     ERROR_INTERRUPT
487    ERROR_INVALID_ADDRESS

```

flags from BSEMEMF.H

```

PAG_READ      0x00000001U
PAG_WRITE     0x00000002U
PAG_EXECUTE   0x00000004U
PAG_GUARD     0x00000008U
PAG_COMMIT    0x00000010U
OBJ_TILE      0x00000040U
PAG_DEFAULT   0x00000400U
OBJ_GETTABLE  0x00000100U
OBJ_GIVEABLE  0x00000200U
PAG_SHARED    0x00002000U
PAG_FREE      0x00004000U
PAG_BASE      0x00010000U

```

```

Event 3 - Allocate 0x200 byte Array - Address 0x00480000 - flags 0x00000057
Event 4 - Allocate 0x200 byte Array - Address 0x00490000 - flags 0x00000057
Event 6 - Allocate 0x200 byte Array - Address 0x004A0000 - flags 0x00000053
Event 7 - Allocate 0x200 byte Array - Address 0x004B0000 - flags 0x00000013
Event 8 - Allocate 0x200 byte Array - Address 0x004C0000 - flags 0x00000003
Query Base 0x00480000 Query Length 0x00001000 QueryFlags 0x00010017
Query Base 0x00481000 Query Length 0x0000F000 QueryFlags 0x00000007
Query Base 0x00490000 Query Length 0x00001000 QueryFlags 0x00010017
Query Base 0x00491000 Query Length 0x0000F000 QueryFlags 0x00000007
Query Base 0x004A0000 Query Length 0x00001000 QueryFlags 0x00010013
Query Base 0x004A1000 Query Length 0x0000F000 QueryFlags 0x00000003
Query Base 0x004B0000 Query Length 0x00001000 QueryFlags 0x00010013
Query Base 0x004B1000 Query Length 0x0000F000 QueryFlags 0x00000003
Query Base 0x004C0000 Query Length 0x00010000 QueryFlags 0x00010003
Hit any key to Continue

```

NOTES:

The query call allows you to look at any address in the linear address space. The second parameter is an input-output parameter. On input, it represents how far you would like to query. On output, it is how far the query proceeded before it found a change in the flags. Certain flags such as OBJ_GIVEABLE, OBJ_GETTABLE, and OBJ_TILE are not reported by this call.

DosSetMem

```

DosSetMem (PVOID pb, <-- region address
          ULONG cb,  <-- number of bytes
          ULONG flag); <-- access requested

```

rc =

| | |
|-------|-------------------------------|
| 0 | NO_ERROR |
| 5 | ERROR_ACCESS_DENIED |
| 8 | ERROR_NOT_ENOUGH_MEMORY |
| 87 | ERROR_INVALID_PARAMETER |
| 95 | ERROR_INTERRUPT |
| 212 | ERROR_LOCKED |
| 487 | ERROR_INVALID_ADDRESS |
| 32798 | ERROR_CROSSES_OBJECT_BOUNDARY |

flags from BSEMEMF.H

| | |
|--------------|-------------|
| PAG_READ | 0x00000001U |
| PAG_WRITE | 0x00000002U |
| PAG_EXECUTE | 0x00000004U |
| PAG_GUARD | 0x00000008U |
| PAG_COMMIT | 0x00000010U |
| PAG_DECOMMIT | 0x00000020U |
| PAG_DEFAULT | 0x00000400U |

NOTES:

This call allows you to set memory attributes on a per-page basis. The attributes you would normally change would be PAG_WRITE, PAG_COMMIT, and PAG_DECOMMIT. Generally you commit memory when you allocate it with DosAllocMem or DosAllocSharedMem. There is seldom a reason not to commit memory when you allocate it now that we have *lazy commit*. That is, backing store is not provided until you touch the memory.

However, if you want to free storage without releasing the memory object, you can decommit it. When PAG_DECOMMIT is the flag parameter, do not use any other flags. It just works that way. If you are a good citizen and decommit memory which you do not need, then you can re-enable the address space by using PAG_COMMIT. PAG_COMMIT can never stand alone as a parameter -- you must use at least one of PAG_READ, PAG_WRITE or PAG_EXECUTE. Alternatively you may use PAG_DEFAULT, which represents the combination of PAG_READ, PAG_WRITE and PAG_EXECUTE attributes you gave the object when you initially allocated it.

PAG_EXECUTE has no effect in the Intel architecture. The only bit in the page tables you can affect is the R/W bit. Thus PAG_READ has no effect. Only PAG_WRITE *does* something. It allows you to make some part of your memory space read-only. Code is always read-only. We will see that when we look at the page tables. Data can be made read-only which is a protection against wild pointers. That is, once you have set up some data and are going to pass that to a subroutine or procedure or DLL, you can make it read only. That will cause an exception should that errant procedure or DLL accidentally try to write to the memory.

Shared memory, once committed, cannot be decommitted. When you change the read/write attributes of shared memory, you are affecting only *your* access to that shared memory.

DosFreeMem (PVOID pb); <-- region address

rc =

| | |
|-----|-----------------------|
| 0 | NO_ERROR |
| 5 | ERROR_ACCESS_DENIED |
| 95 | ERROR_INTERRUPT |
| 487 | ERROR_INVALID_ADDRESS |

NOTES:

DosFreeMem releases a previously allocated private or shared memory object from the virtual-address space of the subject process.

Freeing a shared memory object decrements the reference count for the associated object. If the resulting count is zero (that is, no other references to the shared memory object exist throughout the system), then the object is deleted. The deletion of the shared memory object releases the backing storage for the committed pages within the object.

Concepts of Shared Memory

- 1) **Reference Count** determines Life of Object
- 2) Address Space Reserved for Life of Object
- 3) Page Tables Allocated in other processes when activated
- 4) **Giveable** - You Activate Address Space and Increment Reference Count in context of another process
- 5) **Gettable** - Process Activates its own Address Space and updates its own reference count

| named | unnamed giveable | unnamed gettable |
|--|---|---|
| DosAllocSharedMem | DosAllocSharedMem | DosAllocSharedMem |
| other process activates its own address space using DosGetNamedSharedMem | you activate other process's address space using DosGiveSharedMem | other process activates its own address space using DosGetSharedMem |
| public if you know the name | private | public if you know the linear address |

- Q] Where is the reference count/use count stored? *in Context Record*
- Q] How can we see the reference count with OS/2? Using *use PSTAT* Kernel Debug?

NOTES:

Shared memory is an important form of interprocess communications in OS/2. One strong feature of OS/2 is the isolation of memory objects between processes. Shared memory allows an intentional breakdown of that isolation.

A shared memory object (and a shared semaphore and a DLL) has a life span. It exists as a system object so long as its REFERENCE COUNT is greater than 0. When it is created the reference count is 1. The reference count is always the number of processes which have access to a memory object. Giving a memory object multiple times to the same process has no incremental affect.

The primary benefit of named shared memory is that any process knowing the name can gain access to the memory.

The primary benefit of unnamed giveable shared memory is that a process may increment the reference count of that memory object in the context of a target process, and then free the memory in the context of itself. In this manner, the object can be "transferred" to another process without risk that the reference count will go to zero and the object will be discarded.

DosAllocSharedMem

DosAllocSharedMem (PPVOID ppb, --> pointer to pointer
 PSZ pszName, <-- name or 0
 ULONG cb, <-- size in bytes
 ULONG flag); <-- allocation flags

rc =

| | |
|-----|-------------------------|
| 0 | NO_ERROR |
| 8 | ERROR_NOT_ENOUGH_MEMORY |
| 87 | ERROR_INVALID_PARAMETER |
| 95 | ERROR_INTERRUPT |
| 123 | ERROR_INVALID_NAME |
| 183 | ERROR_ALREADY_EXISTS |

flags from BSEMEMF.H

| | |
|--------------|-------------|
| PAG_READ | 0x00000001U |
| PAG_WRITE | 0x00000002U |
| PAG_EXECUTE | 0x00000004U |
| PAG_COMMIT | 0x00000010U |
| OBJ_TILE | 0x00000040U |
| OBJ_GETTABLE | 0x00000100U |
| OBJ_GIVEABLE | 0x00000200U |

PCH pchArray3;

PSZ pszName = "\\SHAREMEM\\LAB1F";

ulAllocFlags = PAG_READ | PAG_WRITE;

DosAllocSharedMem ((PPVOID) &pchArray3, pszName, 0x20000, ulAllocFlags);

(PSC)

NOTES:

This call is used to allocated named and unnamed shared memory. In fact, the call does not allocate memory but allocates linear address space in the context of this process. The "high" or shared address space is used. Once the address space is assigned, the exact same addresses are reserved in the context of every process which is running and every process which ever will run, until this particular shared memory object is deleted. That happens when the REFERENCE COUNT goes to zero. The reference count is actually the number of context records attached to this particular object's arena record in the shared arena.

If the memory is named, then it is automatically gettable (you must not specify OBJ_GETTABLE). If the memory is unnamed, then you must specify OBJ_GETTABLE or OBJ_GIVEABLE or both. When an appropriate API is executed to activate the address space in the context of another process (DosGiveSharedMem, DosGetSharedMem, or DosGetNamedSharedMem) then the reference count is incremented and the other process may use the address space.

When each process whose address space has been activated frees the address space (DosFreeMem) then the reference count of the shared object goes to zero and the address space is available for reassignment. A process also frees the memory when it terminates (OS/2 does this automatically).

Shared memory objects may be committed at creation or by using DosSetMem. Once committed, shared memory may not be decommitted.

Gettable Shared Memory

Named Shared Memory and OBJ_GETTABLE unnamed shared memory is gettable. That means that any process which wishes to access the memory can issue a `DosGetSharedMem` or `DosGetNamedSharedMem` and then access to the memory object.

`DosGetSharedMem` and `DosGetNamedSharedMem` increment the usage count of that memory object, and activate the address space in the context of the process making the call.

A benefit of gettable objects is that the creator does not have to know the PID(s) of the recipient(s).

A problem with gettable memory objects is that the creator must not free the object before at least one other process gets it, else the object will be discarded.

Unnamed shared memory objects may be both giveable and gettable.

Giveable Shared Memory

Giveable unnamed shared memory may be given to another process using `DosGiveSharedMem`. The effect of this call is to increment the usage count and to activate that linear address space in the donee process. The donor specifies the initial permissions (`PAG_READ`, `PAG_WRITE`) in the donee process.

Giveable objects are private between the processes which share them. Any process holding giveable memory may give it to any other process. Nothing is "lost" when you give shared memory to another process.

As with gettable shared memory, any process having access to the memory object may free it using `DosFreeMem`, or `close` (which has the same effect). This decrements the use count.

DosGetNamedSharedMem

DosGetNamedSharedMem (PPVOID ppb, --> pointer to pointer
 PSZ pszName, <-- name
 ULONG flag); <-- access requested

rc =

| | |
|-----|-------------------------|
| 0 | NO_ERROR |
| 2 | ERROR_FILE_NOT_FOUND |
| 8 | ERROR_NOT_ENOUGH_MEMORY |
| 87 | ERROR_INVALID_PARAMETER |
| 95 | ERROR_INTERRUPT |
| 123 | ERROR_INVALID_NAME |
| 212 | ERROR_LOCKED |

flags from BSEMEMF.H

| | |
|-----------|-------------|
| PAG_READ | 0x00000001U |
| PAG_WRITE | 0x00000002U |

Enables access to a named shared memory object that was allocated by DosAllocSharedMem

Increments the object's reference count

At least one process must call this function prior to the allocating process calling DosFreeMem, else the memory object will be released

```
#define MEMFLAGS PAG_READ | PAG_WRITE
#define MEMNAME = "\\SHAREMEM\\EventLog"
DosGetNamedSharedMem ((PPVOID) &pbBaseAddress, MEMNAME, MEMFLAGS);
```

NOTES:

Named shared memory is the easiest way to share information between a set of processes which you are writing. Shared memory is compatible between 16 and 32-bit applications, and of course between OS/2 and Presentation Manager programs.

Generally your first application will allocate the memory, and other applications or processes which come in later will use the name which will be defined in a common .H file. The use of a #define (shown above) is more efficient than using a memory variable. The #define is also easily replicated in the .H files of every application.

DosGiveSharedMem

```

DosGiveSharedMem (PVOID pb,      <-- identify by address
                  PID pid,      <-- PID of recipient
                  ULONG flag); <-- recipient access rights

```

rc =

| | |
|-----|-------------------------|
| 0 | NO_ERROR |
| 5 | ERROR_ACCESS_DENIED |
| 8 | ERROR_NOT_ENOUGH_MEMORY |
| 87 | ERROR_INVALID_PARAMETER |
| 95 | ERROR_INTERRUPT |
| 212 | ERROR_LOCKED |
| 303 | ERROR_INVALID_PROCID |
| 487 | ERROR_INVALID_ADDRESS |

flags from BSEMEMF.H

| | |
|-----------|-------------|
| PAG_READ | 0x00000001U |
| PAG_WRITE | 0x00000002U |

Gives another process access to an unnamed memory object allocated "giveable" by DosAllocSharedMem

Increments the object's reference count

NOTES:

Since we are activating the recipient's address space, we set the R/W bits in the page table entries for this address space in the recipient's address space. The recipient can always change their access rights using DosSetMem, but we set up the initial rights.

DosGetSharedMem

DosGetSharedMem (PVOID pb, <-- identify by address
 ULONG flag); <-- access rights

rc =

| | |
|-----|-------------------------|
| 0 | NO_ERROR |
| 5 | ERROR_ACCESS_DENIED |
| 8 | ERROR_NOT_ENOUGH_MEMORY |
| 87 | ERROR_INVALID_PARAMETER |
| 95 | ERROR_INTERRUPT |
| 212 | ERROR_LOCKED |

flags from BSEMEMF.H

| | |
|-----------|-------------|
| PAG_READ | 0x00000001U |
| PAG_WRITE | 0x00000002U |

Enables this process to access an unnamed memory object allocated "gettable" by DosAllocSharedMem

Increments the object's reference count

At least one process must call this function prior to the allocating process calling DosFreeMem, else the memory object will be discarded

NOTES:

5] Why would anyone want to "get" shared memory? When would you use named vs unnamed gettable shared memory?

a] _____

6] What is the usage count or reference count with shared memory. How is that different than with private memory?

a] _____

7] In order to "get" the memory, what must another process know? How does shared memory differ from global variables?

a] _____

8] If a process "gives" shared memory to another process, what must also be given? Will the name alone suffice?

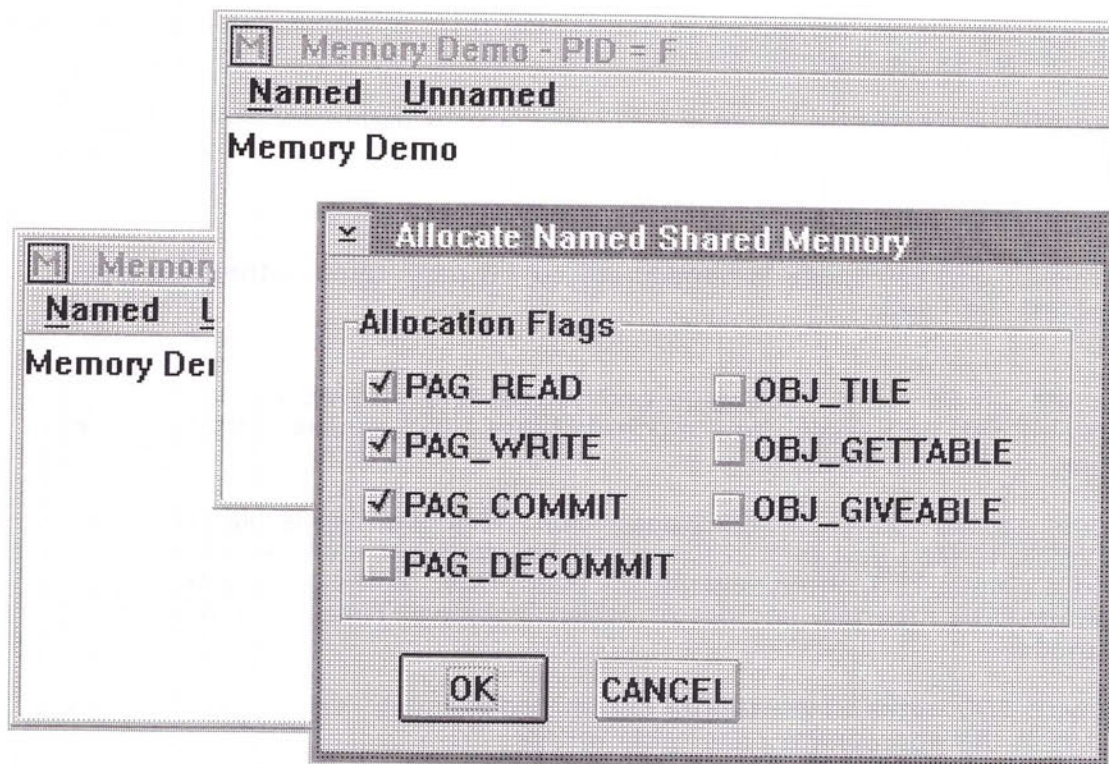
a] _____

9] There are two issues regarding the use of DosSetMem to decommit memory -- one regarding private memory and one regarding shared memory. What are those issues?

a] _____

[] Check your answers on page M-18.

MEMDEM.EXE - Shared Memory Demonstration



NOTES:

Start \OS22LABS\DEMOS\MEMDEM\MEMDEM two or three times. Once named shared memory is allocated, that memory is gettable in the other processes. You may experiment with changing the attributes and freeing the memory. To reallocate you will have to close all the processes.

When unnamed shared memory is allocated, its address is placed where all instances of MEMDEM.EXE can find it. You must allocate it OBJ_GIVEABLE or OBJ_GETTABLE or both. You can then use the Unnamed pull-down to give or get the unnamed shared memory between processes.

Answers to Questions

< page M-5 >

- 1] If I allocate 200 bytes of read/write committed memory, how much memory can I access before I get a protection fault?
 - a] You can access 4K bytes, because OS/2 allocates a page at a time.

- 2] How much page table space is used by the 200 byte allocation?
 - a] 64K of linear address space is consumed, so 16 page table entries (64 bytes of page tables) are used.

- 3] If I allocate 2 64K arrays read/write committed, and they appear in consecutive 64K addresses, how much memory can I access starting at the beginning of the first array?
 - a] You could access 128K without a page fault. This question is placed in the workshop to remind you that in the 16-bit environment each allocation involved a unique LDT entry, and thus a side effect of the segmented architecture was a certain amount of object isolation. That is not the case in flat mode.

- 4] If I allocate 200 bytes of read/write committed memory, when will the present bit be set in the appropriate page table entry?
 - a] The memory will be committed and the present bit will be set when the memory is first accessed. We will be able to see that with Kernel Debug.

< page M-16 >

- 5] Why would anyone want to "get" shared memory? When would you use named vs unnamed gettable shared memory?
 - a] Shared memory is a form of inter-process communication in OS/2. One of the benefits of OS/2 is that processes are isolated from each other. That is also one of the detriments. Shared memory is a way to break down that isolation for selected memory objects.

If you are "posting" information on a "bulletin board", so that any process can get to it, then named shared memory is ideal: any process which knows the name can get access to the memory.

Unnamed shared memory is useful when you have a large number of shared memory objects to pass, and you can conveniently pass the address of the objects to another process via named shared memory or via a queue.

Unnamed giveable memory allows you to pass the memory to another process and then free it. If the memory is gettable, you cannot free it until you know that the other process has executed a "get". In many cases, you are going to keep the memory around for the life of the application, so you do not worry about freeing it. Then gettable memory is fine.

- 6] What is the usage count or reference count with shared memory. How is that different than with private memory?
 - a] There is no reference count for private objects. The reference or use count for shared objects lets OS/2 know when the object can be discarded. The rule is simple: when no one is using it get rid of it.
- 7] In order to "get" the memory, what must another process know? How does shared memory differ from global variables?
 - a] A process can "get" gettable shared memory using the name of the shared memory (if it is named) or the linear address if it is unnamed. The linear address is effectively the "handle" for unnamed shared memory.

Shared memory is different than global variables, in that shared memory can be used between processes while global variables work only in the context of one process.

- 8] If a process "gives" shared memory to another process, what must also be given? Will the name alone suffice?
 - a] Giveable shared memory is always unnamed. Thus the linear address must be communicated. There is no name. It was a trick question.
- 9] There are two issues regarding the use of DosSetMem to decommit memory -- one regarding private memory and one regarding shared memory. What are those issues?
 - a] With DosSetMem when you specify the flag PAG_DECOMMIT, you can specify no other flag bits. It's just the way it works. Shared memory cannot be decommitted.

OS/2 2.1 for Software Developers

Appendix S

Signal Handling

Some material in this chapter
courtesy IBM Toronto Lab

by Charles R. Chernack

Table of Contents

Appendix S - IBM C Set Signal Handling S-1

 Technical Background S-3

 Memory Protect Violation in a PM Program S-5

 Vectoring and Exception S-6

 General Protection and Page Faults S-7

 C Functions for Lab 4 Part A S-8

 Technical Note on Signals S-9

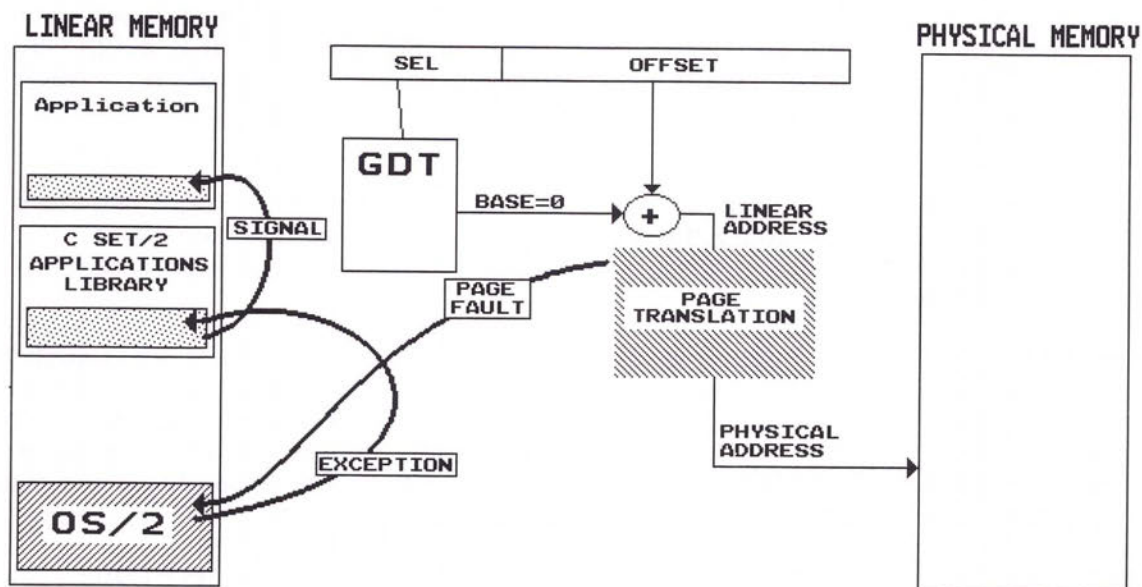
Technical Background

Signals in OS/2 1.X were used for CTRL+C, CTRL+BREAK and SIGTERM notification. Signals were also used as a form of interprocess communication. That is, you could send a signal FLAGA, FLAGB or FLAGC to another process. This caused a *software interrupt* in the context of the target process.

In OS/2 2.1 signals are *not* used as a form of interprocess communication. Generally signals are the C library's way of notifying an application of an event. 32-bit signals work in the context of each thread, and they provide a simplified, standardized and portable way of dealing with extraordinary information.

We will tend to focus on a memory protect violation in this appendix, and also in appendix E (Exception Handling). The two appendices should be read together. But we will broaden the discussion to include other extraordinary events.

Signals are a much simpler and less powerful way of dealing with exceptional events which occur in the context of a thread. But in many cases signal handling will provide the tools you need to get your job done. They are not as robust as exception handlers, but they may get your job done.



In the figure above an application accesses address space based on the Intel hardware: there is a 448 MB flat segment (0:32 segment) described in the Global Descriptor Table, and then there are specific pages in that virtual 448 MB which are mapped to physical memory via Intel hardware page translation. The GDT and the page translation tables are set up by the operating system. The SEL portion of the address is set up for you by the OS/2 loader to point to the correct GDT entry (53H).

The program provides a virtual (linear) address using a ULONG pointer reference. For example, if the program references address 0x100, it becomes 53H:00000100H. The 100H (hex) is technically the OFFSET from the start of the segment. The segment base is equal to 0, so the linear (virtual) address is 0x100. If the linear address is greater than 448 MB (the GDT limit) there is a general protection fault. If the linear address is mapped to a not present page, there is a page fault. In the case of 0x100, there would be a page fault since OS/2 never assigns addresses below 0x10000 to the application.

The page fault causes a hardware interrupt which is routed to the OS/2 kernel. Now not all page faults are fatal. You might reference a page which has been assigned to you but for which there is currently no backing store. OS/2 will then handle the page fault by allocating the storage, and then go back into your application and re-run the offending instruction (this time, without fault).

If you get a GP fault (address > 448 MB) or you get a fatal page fault, then OS/2 will put up the hard error screen and terminate your application. You can use AUTOFAIL and/or DosError to suppress the hard error screen.

If the thread which caused the fault has registered an exception handler (see Appendix E), then before taking the default action (of terminating your application) the OS/2 kernel will call your exception handler.

If you use /Rn when you compile, then you are using the **subsystem** library which does not contain exception handlers. We shall assume that you did not use /Rn. In that case, the C Set runtime registers its own exception handler **_Exception** for each thread you create using **_beginthread**. The C startup code registers this handler for thread 1.

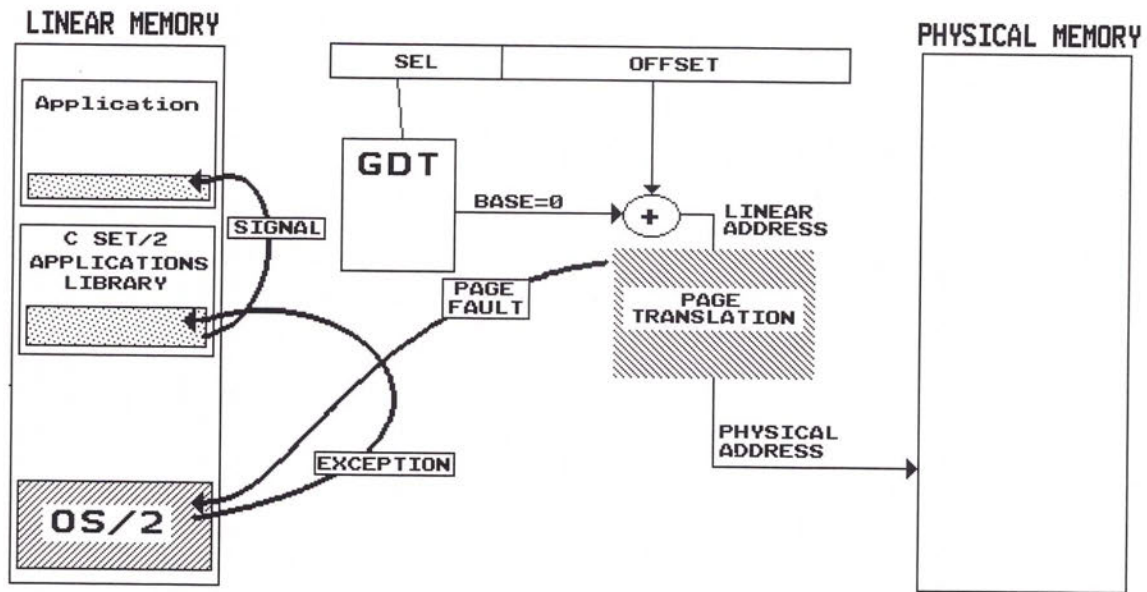
The illustration on page DLL-18 shows the startup code. As you can see, it registers **_Exception** for thread 1. The primary difference between **DosCreateThread** and **_beginthread** is that **_beginthread** registers **_Exception** for each thread you create. There are other differences too...

A flowchart of C's **_Exception** handler is shown on page E-9. This flowchart is slightly different for C Set ++ and C Set/2: on a fatal error C Set/2 will do a register dump and terminate the application, while C Set ++ will do a register dump and then go back to OS/2 and let OS/2 terminate the application. That way you will see the hard error screen. The register dump to stdout does not appear in Presentation Manager programs, but the hard error screen does. That was the reason for the change.

The C **_Exception** handler will call your signal handler if you have registered one. Signal handlers are not called for *every* exception -- so to be truly robust you would register your own exception handler. Signal handlers are *much* simpler to write, as the C compiler does the hard work.

Signal handling across DLL boundaries does not work, unless you have one and only one copy of the runtime linked to your application. This is the issue of different library environments. If a DLL has statically linked the runtime you will not process signals registered outside that DLL. See **#pragma handler** in appendix E.

Memory Protect Violation in a PM Program



```
#include <signal.h>
static void signalhandler(int iSignalNumber); // signal handler prototype

signal (SIGSEGV, signalhandler); // register the signal handler for this thread

static void signalhandler(int iSignalNumber) // put up a message box
{
    DosBeep (1000,60);
    WinMessageBox (HWND_DESKTOP, hwndFrame, "Memory Protect Violation",
    "Signal", ID_MESSAGEBOX1, MB_OK | MB_CUAWARNING | MB_MOVEABLE );
}
```

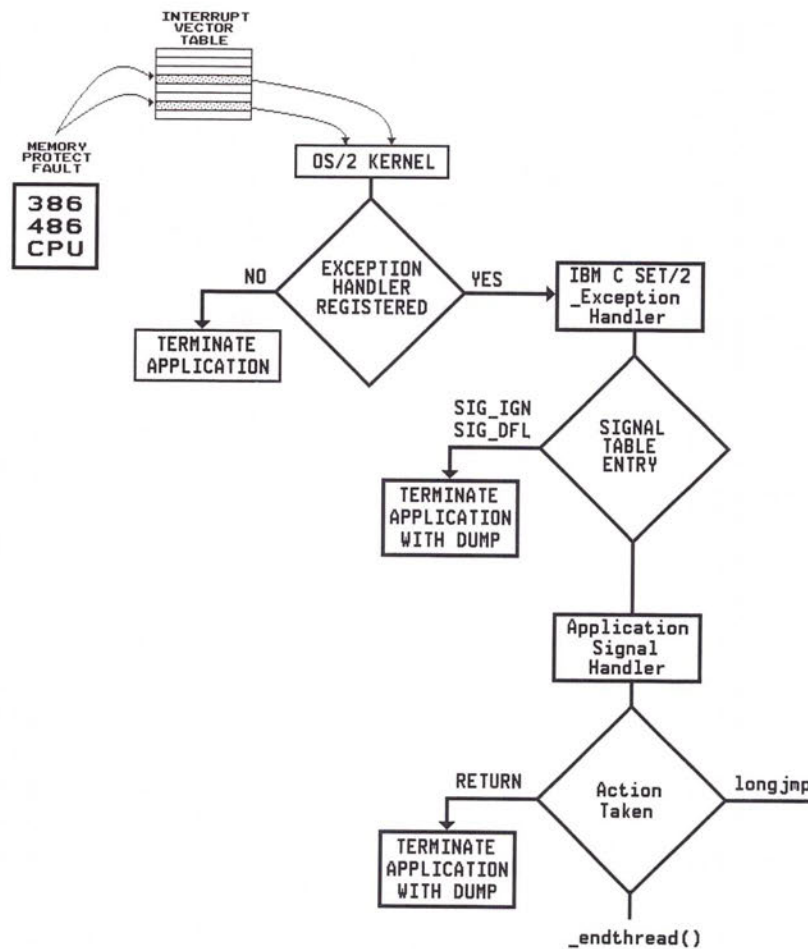
The IBM C Set/2 Applications Libraries install the C exception handler `_Exception` for each thread of your application when you use the `/Gm` option and `_beginthread` to create the new threads. If you use the `/Gm-` option then the handler is installed only for thread one.

If your program, or a thread in your program, generates a memory protect violation, either by referencing an address greater than 512 MB or by referencing a page which has not been assigned to your application, then OS/2 2.1 will call the thread specific exception handler.

The IBM C Set/2 Exception Handler will react to a memory protect violation by (1) calling your signal handler, (2) taking the default action `SIG_DFL`, or (3) attempt to ignore the action `SIG_IGN`. The library maintains a signal table for each thread, for each of 10 signals. The contents of the table can be (1) the address of your handler, (2) `SIG_DFL`, or (3) `SIG_IGN`. In the case of a memory protect violation the default and ignore case will cause your application to be terminated.

In the example above, which is from `\OS22LABS\DEMOS\MEMDEM\MEMDEM.C`, a Presentation Manager program registers a handler which will put up a message box when there is a memory protection violation. This will prevent the application from just "disappearing" from the screen. Since the handler returns, the application is terminated when the message box is dismissed. This message box is not necessary when using the C Set ++ Library as the C Set ++ `_Exception` handler returns to OS/2 and OS/2 will put up the hard error screen

6 Vectoring of the Exception

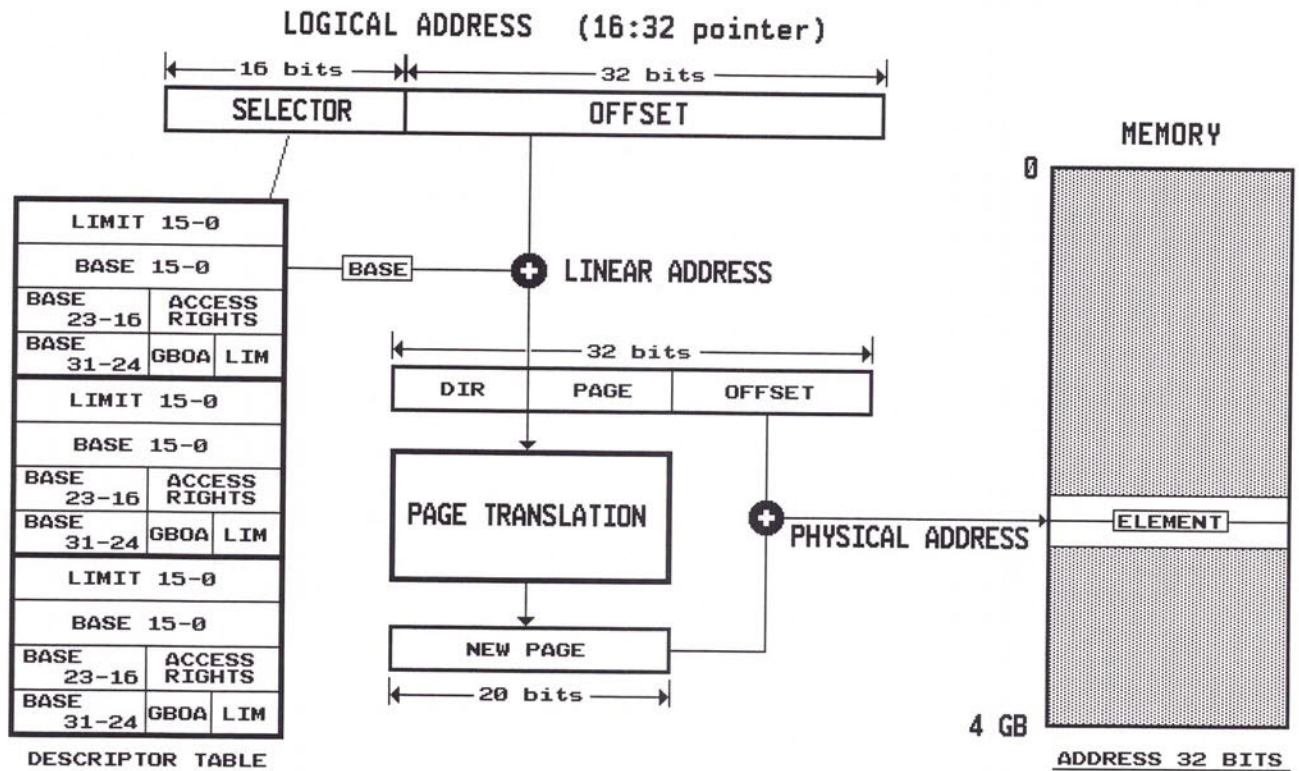


The hardware generates an exception (an interrupt, if you will) if you exceed the GDT limit or reference a not-present page, or try to write to a read-only page. The interrupt vectors through the interrupt vector table (technically the interrupt descriptor table) and ends up in OS/2. The operating system cures the problem if it occurs because one of your pages was swapped out. Else, the operating system will call the thread's exception handler if one has been registered.

The IBM C Set/2 Library registers its `_Exception` handler. That handler has a signal table for each of 10 signals for each thread. If the `_Exception` handler sees a memory protect violation, it then converts it into `SIGSEGV`. The state of the signal table can be `SIG_IGN`, `SIG_DFL`, or "your handler". `_Exception` can not ignore a real memory protect violation, so the ignore and default cases cause your process to be terminated: the C Set/2 `_Exception` handler does a `DosExit (EXIT_PROCESS ...` with a register dump; the C Set ++ `_Exception` handler does the register dump and returns to the OS/2 Kernel.

If you have registered a signal handler, your handler is called. If your handler returns, `_Exception` takes the `SIG_IGN` case (which for a memory protect terminates the process). If your handler does a `longjmp` or it terminates the offending thread (and in either case does not return), your process survives.

General Protection and Page Faults



NOTES:

When the application is running, OS/2 is not. Only the hardware is minding the store. If the application makes a memory reference beyond the GDT limit, it will not get past the first "+" -- there will be a general protection fault. If the application accesses a not present page or attempts to write to a read-only page, there will be a page fault -- it will not get past the second "+".

To review, when a thread accesses invalid memory, the CPU will generate a page fault if the access is within the GDT limit. A general protection fault is generated if the reference is above the GDT limit (448 MB).

If a page fault occurs, OS/2 checks to see that the memory was allocated to the application and the present bit is clear. If so, the memory is loaded from disk and the present bit is set. Otherwise, if an exception handler has been registered, OS/2 raises an exception in the context of the thread.

If a general protection fault occurs, and an exception handler has been registered, then OS/2 will raise the exception handler in the context of the thread.

C Functions for Lab 4 Part A

In the advanced class workshop, labs 4A and 4B let you practice with signal handling. These are the functions we use in those labs:

```
int raise(int);

#define SIGILL      1  <- illegal instruction
#define SIGSEGV    2  <- invalid access to memory
#define SIGFPE     3  <- floating point exception
#define SIGTERM    4  <- OS/2 SIGTERM (killprocess) signal
#define SIGABRT    5  <- abort() signal
#define SIGINT     6  <- OS/2 SIGINTR signal
#define SIGUSR1    7  <- user exception
#define SIGUSR2    8  <- user exception
#define SIGUSR3    9  <- user exception
#define SIGBREAK  10  <- OS/2 Ctrl-Break sequence
```

sends the signal specified to the current thread. The return value is 0 if successful, nonzero if unsuccessful.

```
signal (SIGBREAK, SIG_DFL)    <- reset to default handler
signal (SIGBREAK, SIG_IGN)    <- ignore the signal
signal (SIGBREAK, handler)    <- register handler
```

Allows a process to choose one of several ways to handle an interrupt signal from the Operating System or from the raise function.

The action taken depends on the value of the **func**:

```
#define SIG_ERR ((void (*)(int))-1) <- bad return code
```

```
int atexit (handler);          -> 0 = successful
```

records a function for the system to call at normal program termination. You may register up to 32 functions.

```
void abort (void);
```

similar to exit(), except that exit flushes buffers and closes open files before ending a program. Calls to abort raise the SIGABRT signal.

```
void exit (int status);        <- pass in EXIT_ value 0-255
```

```
#define EXIT_SUCCESS 0
#define EXIT_FAILURE 8
```

returns control to the host environment from the program. exit first calls all functions that the atexit function has placed in a sequential list of functions, in reverse order.

```
APIRET DosError (ULONG error); <- FERR_ flags
```

```
FERR_DISABLEHARDERR    0L <-disable hard error popups
FERR_ENABLEHARDERR     1L <-enable hard error popups
FERR_ENABLEEXCEPTION   0L <-enable exception popups
FERR_DISABLEEXCEPTION  2L <-disable exception popups
```

DosError disables or enables end-user notification of hard errors, program exceptions, or untrapped, numeric processor exceptions. If DosError is not issued, user notification for hard errors and exceptions is enabled.

Technical Note on Signals

The default handling of signals is described in the IBM C Set/2 Language Manual:

| Signal | Source | Default Action | Ignore |
|----------|--|---|-----------|
| SIGABRT | Abnormal termination signal sent by abort function | Terminate the program with exit code 3. | Ignore |
| SIGBREAK | CTRL+BREAK signal | Terminate the program with exit code 99. | Ignore |
| SIGFPE | Floating-Point Exceptions that are not masked, such as overflow, division by zero, and invalid operation | Terminate the program with exit code 99. A machine-state dump is provided. | Terminate |
| SIGILL | Disallowed Instruction | Terminate the program with exit code 99. A machine-state dump is provided unless raise was used to report the signal. | Terminate |
| SIGINT | CTRL+C signal. | Terminate the program with exit code 99. | Ignore |
| SIGSEGV | Access of memory not valid. | Terminate the program with exit code 99. A machine-state dump is provided. | Terminate |
| SIGTERM | Program termination signal sent by the user. | Terminate the program with exit code 99. | Ignore |
| SIGUSR1 | User-defined signals | Ignore | Ignore |
| SIGUSR2 | | | |
| SIGUSR3 | | | |

NOTE: HANDLE is always the same as IGNORE, if you return from the handler.

32-Bit Signals are a creature of the C Set/2 Compiler, and should not be confused with the OS/2 1.X Signaling Capability. The FLAGA, FLAGB, FLAGC form of Interprocess Communication which existed in 1.X does not exist in 2.0.

When the OS/2 2.1 operating system detects something amiss, such as a memory access violation, a math processor error, or a need to terminate an application due to CTRL+C, CTRL+BREAK, or a DosKillProcess, it sends an **exception** to the process.

The chart above shows you all of the C signals. If you do not register a signal handler, and do not set the signal to SIG_IGN, the default action is taken.

SIGBREAK and SIGINT only occur for the main thread and need not even be registered on other threads (save some code). These signals come from the asynchronous exceptions generated by CTRL+C and CTRL+BREAK, or from DosSendSignalException.

Signals and Exceptions on a Per Thread Basis

RULE: Each thread has its own signal handler table. A signal is a creature of the C compiler generated by the `raise()` function. The signal occurs in the context of the thread in which the `raise` was executed. The compiler does this, in part, because in 32-Bit OS/2 the operating system raises exceptions on a per-thread basis.

Mapping Between System Exceptions and C Signals

Some exceptions from the operating system do not have corresponding signals from the library. On the other hand, `SIGABRT` comes from the library function `abort` and does not map into a system exception.

Stack exceptions are not handled by the C library -- they are not converted into signals. The theory behind that is that if you get a stack exception, your C program is already "hosed". However, the option to terminate the offending thread and countine the program argues in favor of future handling of stack exceptions by the library.

Signal Handlers - Registering and Re-Registering

When you start your application, all signals are set to `SIG_DFL` (default action). To specify specific handling for a signal, use the `signal()` call as follows:

| | |
|---|---------------------------------|
| <code>signal (SIGBREAK, SIG_IGN)</code> | <- ignore SIGBREAK |
| <code>signal (SIGABORT, handler)</code> | <- register procedure "handler" |
| <code>signal (SIGILL, SIG_DFL)</code> | <- default handling |

RULE: When you register a signal handler, you are registering it for one specific signal for one specific thread. If you want to register a handler for all threads for all signals, you must call it 10 times (ten signals) inside of each thread! We will do that in lab.

Processing a Signal

If you have registered a signal handler, the handler is called with the signal as a parameter, in the context of the thread where the signal occurred. Thus one handler can handle all signals for all threads, running in the context of the thread in which the signal occurred.

ANSI requires that the handler automatically be de-registered when a signal occurs.

RULE: Inside the handler, re-register the handler for that signal (for that thread).

Why Use Signals - Why Not Exceptions

Signals are easier to use and are portable to other operating environments.

OS/2 2.1 for Software Developers

Appendix STAC

Using Stacker for OS/2

by Charles R. Chernack

Table of Contents

| | |
|--|--------|
| Appendix STAC - Using Stacker for OS/2 | STAC-1 |
| Why a Chapter on Stacker | STAC-3 |
| Installation of DOS and Stacker | STAC-4 |
| Repartition your Hard Disk using FDISK | STAC-4 |
| Stack Drive E | STAC-5 |
| Modify CONFIG.SYS, AUTOEXEC.BAT | STAC-6 |
| Install OS/2 and Enable Stacker | STAC-7 |

NOTE: This Appendix represents personal opinion of the Author and does not represent views held by IBM

Why a Chapter on Stacker

I travel and teach for a living. I carry two OS/2 systems with me -- one using a notebook and one using a sub-notebook. Disk space is very important to me and I decided to try stacker. If you follow their installation instructions, you end up with a system which I found to be hard to use and potentially unstable. That's because (1) they boot off the stacked disk, and (2) I am always changing my config.sys and stacker does *things* when you do that. Also, some applications seem to be sensitive to running in a stacked partition. None of this is hard fact, just opinion and experience. So I figured out how to install stacker to make it pretty bullet proof.

The assumption I make is that you have a system with 120 - 200 MB of hard disk, a fast 386 or 486, and 8 - 16 MB of RAM. You use various communications packages and perhaps various modems including perhaps a PCMCIA modem.

STAC tells you to make a special diskette to boot a stacked copy of OS/2, and they go through serious issues on how to update or install OS/2 using stacker. None of this is necessary.

When you are done with this process, you will have an OS/2 system with much more disk space, and you will be able to reliably boot and you will be able to stack floppy diskettes (PC DOS only). And, unlike the rather confusing environment you get when you follow the STAC installation instructions, you will have a simple and understandable environment.

RULE 1: NEVER depend on STACKER to boot. This does not mean that it does not work, or that there are flaws in their product. Just do not do it.

REASON: If something minor happens to your stacked drive, you may not be able to boot from the stacked drive. But if you can boot without using the stacked drive, it is highly likely that you can repair the stacked drive and not lose data. You do this with the CHECK program.

RULE 2: Always put BOOT MANAGER and PC DOS 6.1 and OS/2 2.1 on the system. That does not mean that you cannot use other versions of DOS, but I happen to like PC DOS 6.1.

REASON: You now have two independent ways of booting. Each way can access the stacked drive. This is an insurance policy. I also like to use the DOS package INTERLNK/INTERSVR to copy files between systems.

RULE 3: Use FAT partitions for OS/2 and PC DOS 6.1. That is not to say that HPFS does not work, but it allows you to tune your OS/2 system when you have booted DOS.

REASON: You can edit and use the DOS partition files from OS/2 and vice versa. You can access at all times the DOS CONFIG.SYS and OS/2 CONFIG.SYS, the DOS AUTOEXEC.BAT and the OS/2 AUTOEXEC.BAT. These DOS files will be in C:\ and the OS/2 files will be in D:\

The installation procedure follows.

Repartition your Hard Disk using FDISK

Use INTERLNK/INTERSVR from DOS, or connect an external SCSI (perhaps using a Trantor interface) and copy everything of value to another system or to an external SCSI. These installation instructions are not for people short on hardware.

Create a 15-20 MB C partition for DOS. We will use this for a minimal DOS (we can keep most of DOS stacked), and for SWAPPER.DAT. We do not want SWAPPER.DAT to be on a stacked disk, and it is good to keep it out of the way of OS/2. Your SWAPPER.DAT might start out at 2 MB, but typically it will need to grow upwards of 10 MB.

Create a 20-25 MB extended DOS partition (D drive) for OS/2 2.1. I prefer 20 MB as that is the smallest partition you can sneak by the OS/2 installation program, but the installation program has a bit of trouble here and 25 MB would likely make it easier. But that extra 5 MB wastes 10 MB of potential stacked disk space. So I used 20 MB.

Create a large extended DOS partition (E drive) using all remaining storage on your hard disk, but for one free track which we will reserve for boot manager.

Install PC DOS 6.1 on Drive C

Boot a DOS 6.1 floppy and **FORMAT C: /S**. Label C: as DRIVE C as you will end up with lots of drives and it is nice to really know which one you are on. Format E: and label it as DRIVE E.

Make a subdirectory C:\DOS and put the following files in it. The rest of DOS can be stacked. If you have any other .SYS files that you use (or any other files you *need* to boot DOS, put them in here or someplace on drive c.

| | |
|----------|-----|
| COMMAND | COM |
| EMM386 | EXE |
| FDISK | COM |
| FORMAT | COM |
| HIMEM | SYS |
| INTERLNK | EXE |
| INTERSVR | EXE |
| MOUSE | COM |
| RAMDRIVE | SYS |
| SETVER | EXE |
| XCOPY | EXE |

You will note that I do not have the DOSSHELL in here. That is because I do not use it. But for your system, put in whatever *you* need to boot DOS.

Make a subdirectory C:\STACKER and put the files below into it from your STACKER installation diskette. We will not use them all, but they are small.

| | | |
|----------|-----|---|
| CHECK | EXE | <- used to verify and repair the stacked disk |
| CREATE | COM | <- used to create stacked drives and diskettes |
| FATMGR | EXE | <- used to tell OS/2 how much space is available |
| REPORT | EXE | <- used to report the status of the stacked disk |
| SDEFRAG | COM | <- used to defragment and optimize the stacked disk |
| SDEFRAG2 | EXE | <- OS/2 version of defragment/optimize |
| SDIR | EXE | <- used to show stacking of a directory |
| SSWAP | COM | <- used to rearrange drive letters under DOS |
| SSWAP | CFG | |
| SSWAP2 | SYS | <- used to rearrange drive letters under OS/2 |
| STACKER | SYS | <- OS/2 stacker driver |
| STACKER | COM | <- DOS stacker driver |
| SWAPMAP | EXE | <- lists all drives and explains which are swapped |

NOTE: You need to install stacker using their standard installation procedure to get some of these files unpacked. You cannot just copy them all from the stacker installation diskette.

Create a subdirectory C:\UTIL or whatever, and put your favorite text editor and other programs which are fundamental to your use of your computer, such as PKUNZIP2 and PKZIP2.

If you are using the TRANTOR SCSI interface, make C:\TSCSI and put in the files your normally use:

| | |
|-------|-----|
| MA348 | SYS |
| TSCSI | SYS |

Stack Drive E

Run **CREATE E:** which will stack drive E: You will not be able to use this disk until you modify your config.sys and reboot. Note that **stacker.com** does not need to be in your config.sys for you to *create* a stacked partition.

Modify CONFIG.SYS, AUTOEXEC.BAT, Reboot

Use your editor to make the following CONFIG.SYS and AUTOEXEC.BAT on DRIVE C:

AUTOEXEC.BAT

```
path c:\dos;e:\dos;e:\util;c:\util;c:\stacker
mouse
doskey
check e: /WP /B
swapmap
```

CONFIG.SYS

```
DEVICE= C:\DOS\HIMEM.SYS
DOS=HIGH,UMB
DEVICE=C:\DOS\SETVER.EXE
DEVICE=C:\DOS\EMM386.EXE          X=D000-DFFF

device = \stacker\stacker.com /EMS e:\stacvol.dsk A:
device=c:\stacker\sswap.com e:\stacvol.dsk

FILES=30
BUFFERS=10
FCBS=4,0
REM DEVICE=C:\DOS\RAMDRIVE.SYS    4096 512 256 /e

rem this line for TRANTOR 348
REM Device=C:\TSCSI\MA348.SYS

rem this line for EXTERNAL SCSI HARD DISK
REM Device=C:\TSCSI\TSCSI.SYS

STACKS=9,256
SHELL=C:\DOS\COMMAND.COM C:\DOS /P /E:240
REM DEVICE=C:\DOS\INTERLNK.EXE /drives:6
```

The A: hanging off the end of stacker.com allows you to access a stacked floppy from DOS. You use **create a:** to stack your floppy. It's a great way to backup files -- no one else will know how to read the floppy!

You should now have a functional bootable DOS system with a large stacked E drive. You can use **REPORT E:** to see what your E drive looks like. Copy everything you own onto this large drive. Try **report E:**. Consider using **sdefrag /GL** to optimize the disk and to adjust your predicted compression ratio. This will take 30-60 minutes on a 150/300 MB partition. Remember to put all of the DOS components you do not need to boot DOS into E:\DOS.

Make a directory C:\OS2\SYSTEM. We will use that for the OS/2 SWAPPER.DAT.

Install OS/2 on Drive D:

You should now install OS/2 2.1 on Drive D:, and also install Boot Manager on that free cylinder. The fact that Drive C is DOS and drive E is stacked has nothing to do with the install process. Do a minimum bootable installation, rather than a custom installation, as there seems to be some problems with custom installation on a 20 MB partition.

You will have to use the FDISK built into Install to make the Boot Manager partition bootable and drive D: installable. Drive D needs to be formatted as a FAT partition.

During the install you can select **software options** and select C:\OS2\SYSTEM for your swappath. It may take multiple passes to install OS/2 2.1 in a 20 MB partition -- I suggest that you install a minimum system at first (select NO options) and then do another pass to add what you want -- after you change the SWAPPATH in CONFIG.SYS.

After the install Boot DOS. Edit D:\CONFIG.SYS to make sure that the SWAPPATH statement says **SWAPPATH=C:\OS2\SYSTEM**. If there is a file D:\OS2\SYSTEM\SWAPPER.DAT remove the file while in DOS.

Now let's see if we can reference the stacked disk from OS/2. Note that nothing required to boot OS/2 is in drive E -- it is all on drive D. Use your text editor to add the following statements to D:\CONFIG.SYS (near the end):

```
device = c:\stacker\stacker.sys e:\stacvol.dsk
device = c:\stacker\sswap2.sys e:\stacvol.dsk
run = c:\stacker\fatmgr.exe
```

Reboot OS/2 using boot manager. You should be able to access your files on Drive E: You can now do a selective install under SYSTEM SETUP and put WINOS2 on Drive E:, or you can install Microsoft Windows on Drive E:. Put your toolkit, compiler, etc on drive E. However, if you are using the execution trace analyzer, you need to have the .SYS file on drive C or D (on a drive which is not stacked):

```
DEVICE=D:\IBMCPPI\SYS\DDE4XTRA.SYS
```

So copy DDE4XTRA.SYS from Drive E: and put it someplace on C or D (I make a directory which has the same pathname as used in E: and just put the .SYS file there).

Everything should now be working. You can boot either DOS or OS/2 regardless of the state of the stacker disk. You have enough room on C or D to put applications which seem to be "stacker sensitive" on a non-stacked disk.

I recommend moving directories not important in the boot process (such as \OS2\HELP) to drive E (and making subsequent changes to D:\OS2\CONFIG.SYS). This frees up valuable uncompressed space. Also, boot DOS and run PC DOS 6.1 DEFRAG on drive C and D (your FAT partitions) now and then. Boot DOS and run SDEFRAG on drive E once in a while. Be careful *not* to run DEFRAG on your stacked disk -- it does not cause data loss but it is a bad idea.

You will find that if you keep many drawings (.BMP, .PCX, etc) zipped to save disk space and that you have to unzip them to use them, that on a stacked drive they might actually take less physical space unzipped!

OS/2 2.1 for Software Developers

Appendix T

Thinking

by Charles R. Chernack

Table of Contents

| | |
|--|------|
| Appendix T - Thunking | T-1 |
| Thunking - Calling Between 32 and 16 Bit Code . . . | T-3 |
| Operation of 16-Bit OS/2 | T-4 |
| Shared Objects in 16-Bit OS/2 | T-5 |
| LDT Tiling vs GDT Enabling: Both = 512 MB | T-6 |
| Tiling the Local Descriptor Table | T-7 |
| 16-Bit Memory Demonstration (LDT in 32-Bit OS/2) . . . | T-8 |
| 16-Bit Processes - Shared Objects in OS/2 2.1 | T-10 |
| The _Seg16 qualifier | T-11 |
| #pragma seg16 - do not cross 64K boundary | T-12 |
| Calling 16-Bit API - Old Style In-Line Thunk | T-13 |
| Compiler Thunking Pathways | T-14 |
| Calling 16-Bit API from 32-Bit C | T-16 |
| EDCThunkProlog, DosFlatToSel, DosSelToFlat | T-18 |
| EDCThunkProlog and #pragma stack16() | T-19 |
| Calling 16-Bit DLL from 16-Bit C | T-20 |
| Calling 16-Bit DLL from 32-Bit C | T-21 |
| 16-Bit Callback to 16-Bit Main | T-22 |
| 16-Bit Callback to 32-Bit Main | T-23 |
| LDT allocation in 16-bit Program | T-24 |
| LDT allocation in 32-bit Program | T-26 |
| Answers to Questions | T- |

Thinking - Calling Between 32 and 16 Bit Code

Operating System Thunks

IBM C Set/2 Compiler Thunks

- Calling 16-Bit OS/2 1.X API
- Calling 16-bit Code in 32-Bit Exe
- Calling 16-bit DLL from 32-Bit Exe
- 16-Bit to 32-Bit Callbacks
- Thinking of Pointers_Seg16
- Preventing Objects From Crossing 64K Boundaries
- Specifying the Stack Size for the 16-Bit Code

Thinking Layers

Thinking refers to calling 16-bit code from 32-bit code, or vice versa. In the Intel chip literature, this is called mixed mode programming. The 80386 and 80486 have the ability to define, in the code segment descriptors, with a single bit, whether or not the code contained in a specific code segment is made up of 32-bit code. Bottom line, the 386 and 486 can "understand" whether the code in any particular code segment is 16 or 32 bits based on the descriptor, and will properly interpret 16 and 32 bit binary codes.

Our job, then, is to understand how we can use both 16 and 32 bit code in the same system. We wish to share memory, semaphores, etc. between 16 and 32 bit processes. We may wish to update a set of 16 bit processes by replacing some with 32-bit processes. We may wish to use a DLL we wrote for a 16-bit application with a 32-bit application. We may wish to provide a thinking layer so that our 32-bit customer may use our old 16-bit DLL transparently.

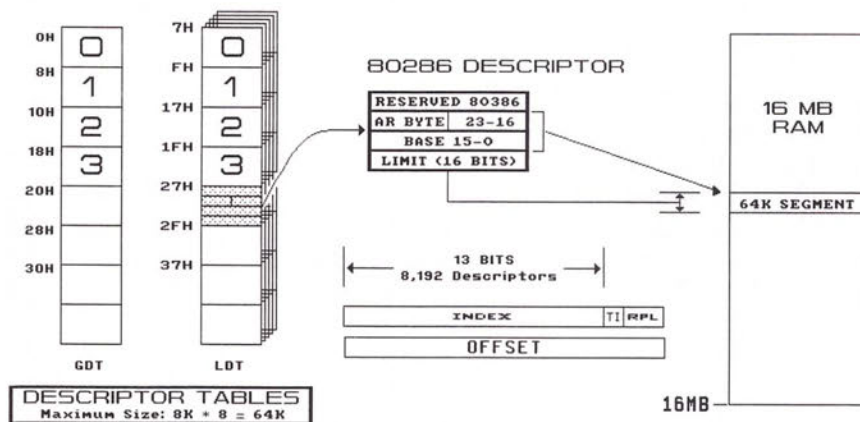
OPERATING SYSTEM THUNKS are required because part of OS/2 2.0 is still 16-bit code. How OS/2 provides 32-bit services to us via 16-bit code is not our problem -- that's an internals issue.

IBM C SET COMPILER THUNKS: The 32-bit compiler has facilities for interconnect to 16-bit API, 16-bit DLLs, and callback from a 16-bit DLL to a 32-bit application. These facilities are built on the ability of the compiler to "package" up a call in the way a 16-bit API or procedure wants to receive the parameters -- using far16 linkage. The compiler can also carry pointers as 16:16 pointers or as 0:32 pointers, and thunk across the assignment operator. The compiler can insure that adequate (or at least user specified amounts of) stack are available in the current 64K of the stack frame when a 16 bit function is called.

The operating system provides Tiled LDTs and simultaneous allocation of memory objects in the 32-bit address space and 16-bit address space, allowing easy coexistence of 16 and 32 bit code in the same process.

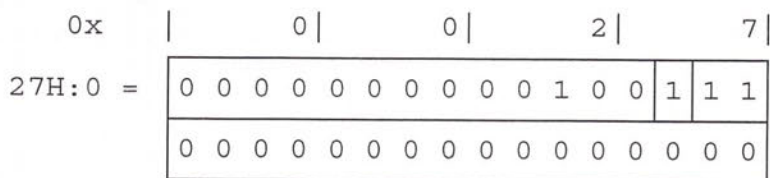
Operation of 16-Bit OS/2

```
DosAllocSeg (0, &sel, SEG_NONSHARED);
pchData = MAKEP (sel, 0);
```



Although you can use the 16 bit API without understanding the mechanics, a basic understanding of the 16-bit mechanics is helpful. In 16-bit OS/2 (and when 16-bit applications run in 32-bit OS/2), application code and data is supported via memory descriptors in the LDT (local descriptor table). These descriptors are created by OS/2 and are understood by the Intel hardware. A descriptor contains the base address and size (LIMIT) of a block of memory that this application can use. A descriptor table is like an array of structures. Each structure has a base, a limit, and an access rights byte. The Intel hardware prevents an application for addressing memory blocks which are not described by its descriptor table. For process isolation, each application has its own descriptor table. They are "local" to the application.

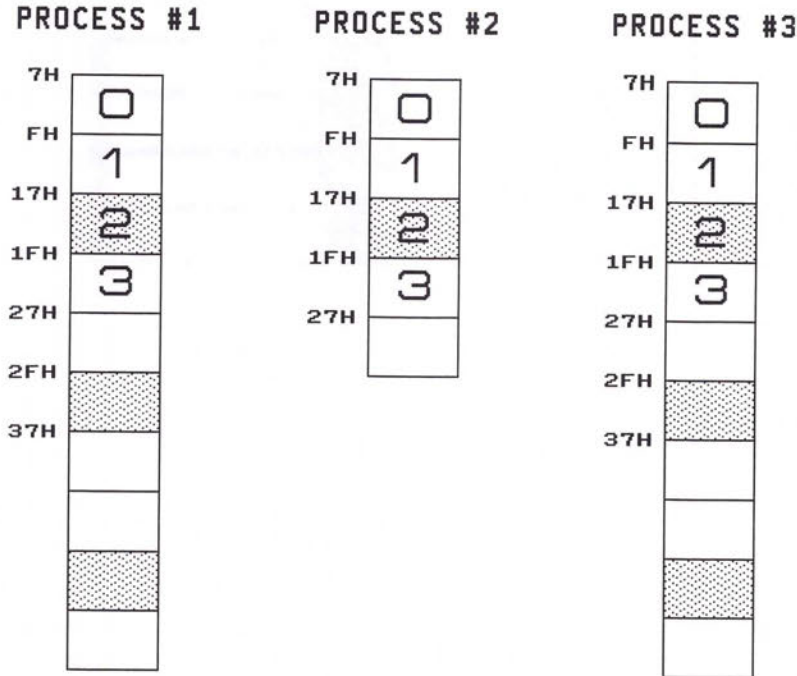
In 16-bit OS/2 a memory address is a SEL:OFFSET pointer made using macro MAKEP. User segments are in LDTs (local descriptor tables) which are specific to each process. LDT entry 4 would have an "index" field of 4, a TI (table indicator) bit of 1 (LDT), and a RPL (privilege level) of 3 (ring 3). The 16:16 pointer to the start of the segment would then be 27H:0.



The left-most 13 bits of the selector index into the descriptor table. The next bit is 0 for the GDT (global descriptor table) or 1 for the LDT.

Shared Objects in 16-Bit OS/2

`DosAllocSeg (0, &sel, SEG_GIVEABLE);`



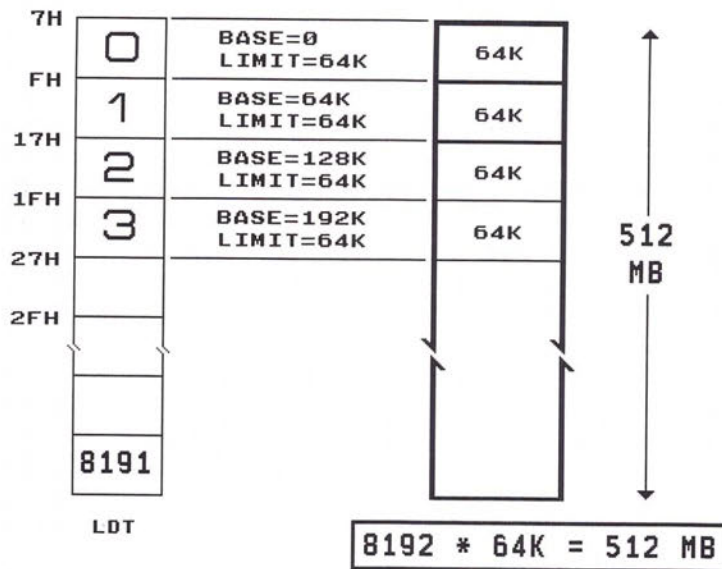
Shared objects are allocated in the DISJOINT space of the LDTs in OS/2 1.X. This allows the LDTs for small processes to remain small, in that both shared and private objects start from the low end of the LDT.

Allocating a shared object reserves an entry in the disjoint space of each processes LDT, but the LDT entry is not activated until the shared memory is enabled by `DosGiveSeg`, `DosGetSeg`, etc.

In 16-bit OS/2 we wish to keep the LDTs small. This is done to save physical memory. Since each shared object must exist at *exactly* the same LDT ordinal in the context of each process which can attach the object. If one process has a large number of private objects, and there were no *disjoint* space, then the first available ordinal for a shared object would be high in the LDT. This would cause the LDT of each application gaining access to that object to be expanded to include that ordinal. The disjoint space -- reserving every few ordinal for shared objects -- solves that problem and is architected into 16-bit OS/2.

When 16-bit applications run in a 32-bit OS/2 environment, their LDTs are organized differently by the system: shared objects start high in the LDT and work toward the center; private objects start low in the LDT and work toward the center.

Tiling the Local Descriptor Table



EXERCISES:

- 1) Explain why selector 0x001F represents LDT ordinal #3.
- 2) Convert 16:16 address 0x1F:0x005 to a 32-Bit Linear Address.
- 3) Convert Linear Address 0x10050 to a 16:16 Address.

If the LDT were shared between all applications and page translation were the sole form of memory management for 16-bit applications running under 32-bit OS/2, then the limit would be 64K in each LDT entry. However, as we will see, there is a separate LDT for each 16-bit application and the LIMIT of each entry is determined by the application (just as in 16-bit OS/2). The LIMIT *could* be 64K but need not be.

The BASE of each successive LDT entry is 64K greater than the previous. Since each 64K boundary represents 0x10000, ordinal 1 is 0x10000, ordinal 2 is 0x20000, etc. OS/2 2.0 loads applications at 0x10000 (which is why we say /BASE:0x10000 using LINK386).

Generally the GDT limit is 448 MB while running the application, as the 448 - 512 MB region is reserved for the protected DLL data area. Thus 1000 LDT entries (64K * 1000 = 64 MB) are not used for normal process objects.

If you realize that each line on the chart represents 0x10000, then the answers to the questions are [1] convert to binary and strip the right three bits -- we did this before; [2] 30000 + 5 = 0x30005; [3] 10000 = F so 0xF:50. If you got these right, you can think addresses!

16-Bit Memory Demonstration

The demonstration program on the next page was written to answer a simple question: when 32-bit OS/2 hosts a 16-bit application does OS/2 create one LDT per application or does it share an LDT for all applications.

The question is interesting because it increases our understanding of *this implementation* of OS/2. Certain programming techniques and observations which we make once we understand the answer to this question will let us intuitively write better code since we know what is happening. But when writing this "better code" we still follow the rules. For example, we know that if you do a DosAllocMem of 200 bytes in OS/2 2.1, that you really get 4K. We would *not* use that information to provide 4K of storage -- as that might change with a different hardware platform. But we *would* use that information to understand a fault at the 4K boundary.

If there were one LDT in 32-bit OS/2 which was shared by all 16-bit applications, then when a 16-bit application did a DosAllocSeg of 200 bytes, it would be able to reference 4K bytes. That would be because the LIMIT field in each entry of the LDT would be set to 0xFFFF and memory would be managed with page translation. That is exactly what happens for 32-bit applications: the first 512 MB (448 MB) is wide open in that the GDT entry has a base of 0 and a limit of 512 MB (448 MB). The same strategy could be used by 32-bit OS/2 for 16-bit applications: make the **one** LDT wide open and manage the memory solely with page translation. That is not the way it is done. Each 16-bit application has its own LDT in OS/2 2.1.

To prove this we run the program on the next page in a 16-bit environment and in a 32-bit environment. The program allocates two 200 byte segments -- a shared segment and a private segment. The shared segment is allocated just so that we can look at its selector. In the case of the private segment, we call an assembler subroutine **usSegmentSize** (source code in the directory) which returns the LIMIT of the LDT entry. If the limit is 0xFFFF we would guess that OS/2 2.1 only needs one LDT for all 16-bit applications. If the limit is 199 then the LDT has been customized for this particular memory allocation and is thus process specific.

In fact, the limit is 199. Also, if we drive a pointer through the private memory we get a fault at 200 rather than at 4096. Again, this shows that we are hitting the LDT descriptor limit. Had this been a 32-bit application, we would have got the fault at 4096 as we would have allocated one page.

In fact, there *is* one page of backing store (4096 bytes on the Intel chip) but you can only access 200 bytes of it.

Note that when we run this application in 16-bit OS/2, the selectors are low in the LDT. That's because shared objects are in the *disjoint* space of the 16-bit OS/2 LDT. When we run this application in 32-bit OS/2, the selectors are at opposite ends of memory because private objects start in low memory and shared objects start in high memory, and they both work towards the center. This is sparse allocation of the address space.

16-Bit Memory Demonstration

```
// demonstrates segment limit of 16-bit program using LSL instruction
// ***** OS/2 Version 1.x ***** in \OS22LABS\DEMOS\MEMORY
```

```
#define INCL_BASE
#include <os2.h>
#include <stdio.h>
#include <stdlib.h>
```

```
#define STDOUT 1
```

```
// internal function prototypes
```

```
void terminate (PCHAR pszMsg, USHORT rc);
```

```
// external function prototypes
```

```
extern USHORT APIENTRY usSegmentSize (SEL sel);
```

```
void main (void)
```

```
{
    USHORT rc;
    USHORT i;
    PCH pch;
    SEL sel;
```

```
rc = DosAllocSeg (200, &sel, SEG_GIVEABLE);
if (rc) terminate ("DosAllocSeg Failure", rc);
printf ("Shared Segment at %4X \n", sel);
```

```
rc = DosAllocSeg (200, &sel, SEG_NONSHARED);
if (rc) terminate ("DosAllocSeg Failure", rc);
printf ("Private Segment at %4X \n", sel);
```

```
printf ("Segment Limit is %d \n", usSegmentSize (sel));
DosSleep(2000);
```

```
for (i = 0; i < 8000; i++) {
    pch = MAKEP (sel, i);
    *pch = 'a';
    printf ("%d \r", i);
}
```

```
DosExit (EXIT_PROCESS, 0);
```

```
}
```

PRINTOUT FROM OS/2 1.3

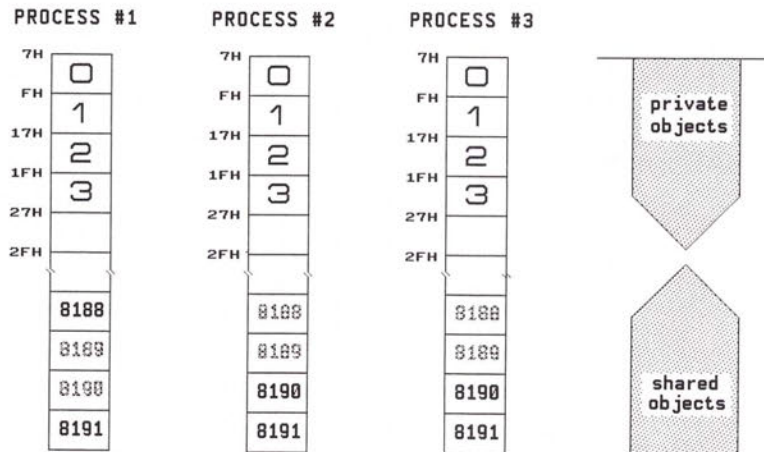
```
Shared Segment at 47F
Private Segment at 127
Segment Limit is 199
```

PRINTOUT FROM OS/2 2.1

```
Shared Segment at B997
Private Segment at 4F
Segment Limit is 199
```


16 Bit Processes: Shared Objects OS/2 2.1

```
DosAllocSeg (0, &sel, SEG_GIVEABLE);
```



Each 16-bit application has its own LDT in 32-bit OS/2. By comparison, all 32-bit applications share a few entries in the GDT. Separate LDTs allow each 16-bit memory object to be allocated with a limit of byte granularity, as in the 1.X environment.

LDTs are allocated as sparse objects in 32-bit OS/2. Allocating a shared object allocates one (or more) entries at the high end of your LDT. Doing this also reserves an entry at the high end of each processes LDT, but their LDT entries are not activated until the shared memory is enabled by `DosGiveSeg`, `DosGetSeg`, etc. That is, the OS/2 rules of memory sharing (give-get and named shared) are always operative.

Why did we use *disjoint* space in 16-bit OS/2? Our objective was to have small LDTs. However, in 32-bit OS/2 the LDTs are populated from both ends. Thus, the *virtual LDT* is always 64K bytes long. However, due to lazy commit, the working ends of the LDT are backed by real storage in RAM or on DASD. The unused middle of the LDT is not backed by RAM or Disk Swap Space until the application uses those LDT entries. Thus there is no cost in having a "big" LDT.

Of course, all of this discussion is implementation oriented. The 16-bit application does not care where its segments are placed in the LDT: the LDT entries are but parking places for segment descriptors.

The `_Seg16` qualifier

```
void main ( void )
{
    APIRET rc;

    PULONG pulLinear;
    PULONG _Seg16 pulSelOffset;

    pulLinear = &rc;
    pulSelOffset = &rc;

    printf ("The linear address is %8X \n", pulLinear);

    printf ("The SEL:OFFSET address is %8X \n", pulSelOffset);
}
```

PRINTOUT:

```
The linear address is    22C48
The SEL:OFFSET address is 172C48
```

You know that in compiler languages, you can carry numbers as fixed point and as floating point, and that the compiler will convert across the assignment operator. The C Set Compiler (which only emits 32-bit code) will carry pointers as 16:16 pointers or 0:32 (flat) pointers. The *type* of the pointer defaults to 0:32, but the `_Seg16` qualifier tells the compiler to carry the pointer as a 16:16 ("thunked") pointer.

The `_Seg16` type qualifier is used when calling 16-bit code to ensure correct mapping of pointers. `_Seg16` is used to create a pointer that can be addressed by a 16-bit program. The pointer can also be used in a 32-bit program, because the compiler converts it to 32-bit form. The `_Seg16` qualifier can only be used with pointers.

Pointers shared between 32-bit and 16-bit code may be qualified with `_Seg16`. This includes pointers passed indirectly to 16-bit code, such as pointers in structures and pointers that are referenced by pointers passed directly to 16-bit code. Pointers passed in function calls are automatically thunked. We'll see that in a subsequent example.

In the example above, `pulLinear` is a 0:32 pointer and `pulSelOffset` is a 16:16 pointer. We take the address of variable `rc`. We then print the address as a 0:32 address and as a 16:16 address. Your knowledge of thunking should show you that flat address 0x20000 maps to 0x17:0, so the thunk is correct.

#pragma seg16 - do not cross 64K boundary

```
#define INCL_BASE
#include <os2.h>

#pragma seg16 (bArray16a) seg16 (bArray16b)

// global arrays

BYTE  bArray32a[60000];      // may cross 64K boundary
BYTE  bArray32b[60000];      // may cross 64K boundary
BYTE  bArray16a[60000];     // will not cross 64K boundary
BYTE  bArray16b[60000];     // will not cross 64K boundary

void main ( void )
{
    printf ("The linear address of bArray32a is %X - %X \n", bArray32a, bArray32a + 59999);
    printf ("The linear address of bArray32b is %X - %X \n\n", bArray32b, bArray32b + 59999);
    printf ("The linear address of bArray16a is %X - %X \n", bArray16a, bArray16a + 59999);
    printf ("The linear address of bArray16b is %X - %X \n", bArray16b, bArray16b + 59999);
    exit(0);
}
```

PRINTOUT:

```
The linear address of bArray32a is 40CE4 - 4F743
The linear address of bArray32b is 4F744 - 5E1A3

The linear address of bArray16a is 20000 - 2EA5F
The linear address of bArray16b is 30000 - 3EA5F
```

The #pragma seg16 directive causes the compiler to lay out the identifier in memory such that it does not cross a 64K boundary. The identifier can then be used in a 16-bit program.

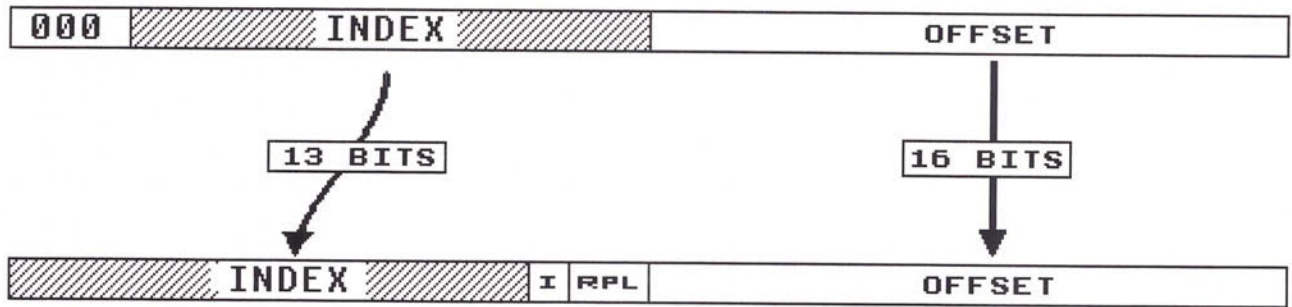
```
>>—#—pragma—seg16—(—identifier—)—<<
```

You can also use the /Gt compile-time option to perform the equivalent of a #pragma seg16 for all variables in the program.

If #pragma seg16 is used on variables of a structure type, the pointers inside that structure are not automatically qualified as usable by 16-bit programs. If you want the pointers in the structure qualified as such, you must declare them using the _Seg16 type qualifier.

In the example above, arrays **bArray32a** and **bArray32b** are permitted to cross 64K boundaries. The printout shows that **bArray32b** does cross a 64K boundary. The second two arrays do not cross a 64K boundary. Thus it would be easy to pass the address of these arrays to a 16-bit DLL. You could not pass the address of these to a 16-bit process (from this 32-bit process) because these memory objects are private. Well, you could pass the address, but the other process could not reference the memory...

Thinking - Calling 16-Bit API



32 Bits = 4 GB
 31 Bits = 2 GB
 30 Bits = 1 GB
 29 Bits = 512 MB

```
void main(void) {
    SEL selL, selG;
    ULONG rc;

    rc=DosGetInfoSeg(&selG,&selL);
}
```

<- Call 16-Bit API from 32-Bit C

```
main PUBLIC main
PROC
PUSH    04H
PUSH    01058H
CALL    EDCThunkProlog
PUSH    EBP
MOV     EBP,ESP
SUB     ESP,010H

SUB     ESP,038H
LEA    EAX,[EBP-04H]; selG
MOV    ECX,EAX

AND    EAX,0ffff0000H
AND    ECX,0ffffH
SAL    EAX,03H
OR     EAX,070000H
OR     EAX,ECX
```

This is assembler code generated by an early (pre-GA) version of the C Set/2 Compiler using the /Fa option

We will discuss this call shortly

Pick up 0:32 address of parameter
 Put it both EAX and ECX

isolate high word of linear address
 isolate low word of linear address
 shift EAX left 3 bits
 and OR in 111 binary
 EAX is now a 16:16 pointer

This is an example of a pre-GA version of C Set/2, where address thinking was done in line. In early beta releases, function prototypes were provided for the 16 bit API (such as DosGetInfoSeg). Here we see the 29-bit linear address of variable segG converted into a 16:16 address in preparation for calling DosGetInfoSeg, a 16-bit API (from a 32-bit application).

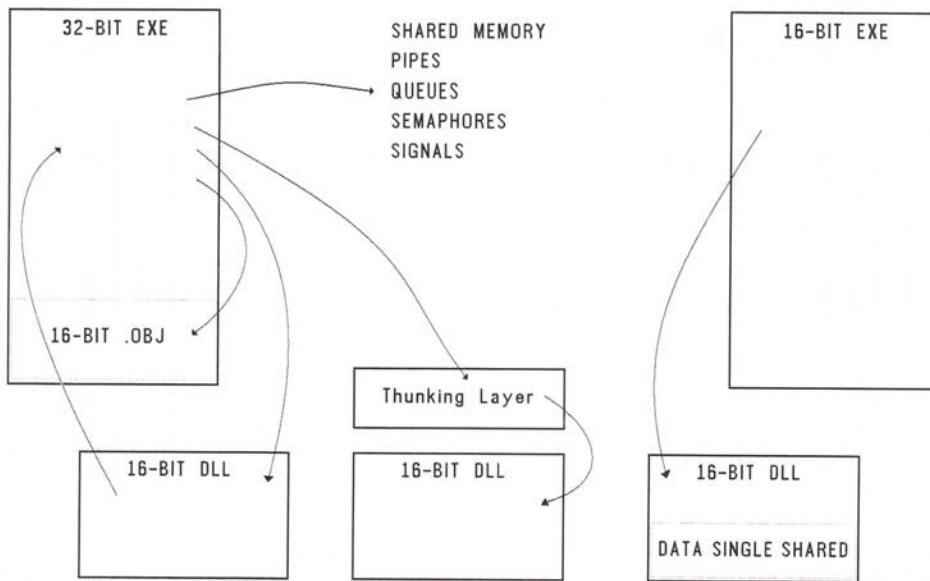
The drawing on top is a 0:32 address. It is a 29 bit address as 29 bits is 512 MB. To think all we have to do is take bits 28-16 (which 0x10000 block) and put them into the 16:16 index field, and take bits 15-0 (offset within 64K) and put them into bits 15-0 of the 16:16 address. Thus a simple bit shift and move thinks the address.

The right-most 16 bits of the linear address becomes the right most bits of the 16:16 address. Those are held in ECX.

The right-most three bits in the upper word are I-RPL. These are set to 111 binary to specify the LDT (I=1) and to specify ring three (RPL=11 binary).

Compiler Thinking Pathways

- DLL USED AS PICKUP TRUCK TO CONTRIBUTE DATA FOR SHARING BETWEEN PROCESSES.



Operating System Thinks

IBM C Set/2 Compiler Thinks

- Calling 16-Bit OS/2 1.X API
- Calling 16-bit Code in 32-Bit Exe
- Calling 16-bit DLL from 32-Bit Exe
- 16-Bit to 32-Bit Callbacks
- Thinking of Pointers _Seg16
- Preventing Objects From Crossing 64K Boundaries
- Specifying the Stack Size for the 16-Bit Code

Thinking Layers

Internal to OS/2, if any of the 32-bit API are supported by old 16-bit code, then thinking is required. We are not concerned with how OS/2 does that. We are concerned with thinks we may wish to invoke ourselves using the IBM C Set Compiler.

We can use thinks to call *any* 16-bit API from our 32-bit application. This makes it easy to share semaphores between a 32-bit application and a pre-existing 16-bit application: we simply use the 16-bit semaphore API. Shared memory is no problem, because it is easily shared between 32 and 16-bit applications.

It is possible to call 16-bit code which has been linked into a 32-bit exe, but practically the linker cannot handle 32-bit C and 16-bit C at the same time. So that capability is useless unless your 16-bit code is in assembler. However, since a 16-bit DLL is separately linked (but runs in your process space), we can and will call 16-bit code in 16-bit DLLs from our 32-bit application. You can also do a *call back*. That is, a 32-bit application can pass the address of an entry point to a 16-bit DLL, and that 16-bit DLL will be able to call back to the 32-bit application. This capability was used in the 16-bit extended services, and it is supported by the 32-bit compiler. We'll look at examples of a 16-16 and 16-32 callbacks.

A thinking layer is a DLL that you can provide to allow your 32-bit customer to call a 16-bit DLL without going through the thinking steps in their code. You put the thinking steps in a 32-bit DLL which you use to front-end a 16-bit DLL.

In the following pages, we'll look at examples of some of these pathways:

| | |
|--|------|
| Calling 16-Bit API from 32-Bit C | T-16 |
| EDCThunkProlog, DosFlatToSel, DosSelToFlat | T-18 |
| EDCThunkProlog and #pragma stack16() | T-19 |
| Calling 16-Bit DLL from 16-Bit C | T-20 |
| Calling 16-Bit DLL from 32-Bit C | T-21 |
| 16-Bit Callback to 16-Bit Main | T-22 |
| 16-Bit Callback to 32-Bit Main | T-23 |

Calling 16 Bit API from 32-Bit C

```
#define INCL_BASE
#include <stdio.h>
#include <os2.h>
#include "getiseg.h"

#pragma stack16(8192)
#pragma linkage (DosGetInfoSeg, far16 pascal)
USHORT DosGetInfoSeg (PSEL selG, PSEL selL);

void main(void) {
    SEL selL, selG;
    PGINFOSEG pginfoseg;
    ULONG rc;

    rc = DosGetInfoSeg(&selG,&selL);

    if (rc) {
        printf("\nReturn %d from DosGetInfoSeg\n",rc);
        return;
    }

    pginfoseg = MAKEP (selG,0); // <- conversion of pointer across assignment operator

    printf("\nThere are %d milliseconds in the timer interval. "
        "\n The day is %d, the month is %d, the year is %d. "
        "\n The major version is %d, and the minor version is %d.\n",

        (ULONG)pginfoseg->cusecTimerInterval,
        (ULONG)pginfoseg->day,
        (ULONG)pginfoseg->month,
        (ULONG)pginfoseg->year,
        (ULONG)pginfoseg->uchMajorVersion,
        (ULONG)pginfoseg->uchMinorVersion);
}
```

The program above is in \OS22LABS\SAMPLES\INFOSEG. It illustrates calling a 16-Bit OS/2 1.3 API from OS/2 2.1. The two statements below are not needed in the program in that the pointer parameters to the 16-bit API will automatically be thunked by the compiler, and that we prefer to carry **pginfoseg** as 0:32.

```
USHORT DosGetInfoSeg (PSEL _Seg16 selG, PSEL _Seg16 selL);
PGINFOSEG _Seg16 pginfoseg;
```

The key statement is the **#pragma linkage** (although it is preferred to put the linkage words right in the function prototype). The pragma linkage or linkage words tell the compiler that this is a 16-bit function which needs to be called using the pascal (system) calling convention.

We have declared a PGINFOSEG (on the next page) in getiseg.h. We get back a selector which is converted to a 16:16 pointer by MAKEP and then converted across the assignment operator to flat pointer **pginfoseg**.

GINFOSEG structure in getiseg.h

```

typedef struct _GINFOSEG {
    ULONG    time;                /* time in seconds          */
    ULONG    msec;               /* milliseconds             */
    UCHAR    hour;              /* hours                    */
    UCHAR    minutes;           /* minutes                  */
    UCHAR    seconds;           /* seconds                  */
    UCHAR    hundredths;        /* hundredths               */
    USHORT   timezone;          /* minutes from UTC         */
    USHORT   cusecTimerInterval; /* timer interval (units = 0.0001 seconds) */
    UCHAR    day;               /* day                      */
    UCHAR    month;             /* month                    */
    USHORT   year;              /* year                     */
    UCHAR    weekday;           /* day of week              */
    UCHAR    uchMajorVersion;    /* major version number     */
    UCHAR    uchMinorVersion;    /* minor version number     */
    UCHAR    chRevisionLetter;   /* revision letter          */
    UCHAR    sgCurrent;          /* current foreground session */
    UCHAR    sgMax;              /* maximum number of sessions */
    UCHAR    cHugeShift;         /* shift count for huge elements */
    UCHAR    fProtectModeOnly;   /* protect mode only indicator */
    USHORT   pidForeground;       /* pid of last process in foreground session */
    UCHAR    fDynamicSched;      /* dynamic variation flag   */
    UCHAR    csecMaxWait;        /* max wait in seconds      */
    USHORT   cmsecMinSlice;       /* minimum timeslice (milliseconds) */
    USHORT   cmsecMaxSlice;       /* maximum timeslice (milliseconds) */
    USHORT   bootdrive;          /* drive from which the system was booted */
    UCHAR    amecRAS[32];         /* system trace major code flag bits */
    UCHAR    csgWindowableVioMax; /* maximum number of VIO windowable sessions */
    UCHAR    csgPMMMax;          /* maximum number of pres. services sessions */
} GINFOSEG;
typedef GINFOSEG FAR *PGINFOSEG;

```

OUTPUT FROM THE PROGRAM:

```

There are 310 milliseconds in the timer interval.
The day is 4, the month is 3, the year is 1993.
The major version is 20, and the minor version is 0.

```

This structure was copied from the OS/2 1.3 Toolkit file BSEDOS.H. The structure was included in the program on the previous page with the #include "getiseg.h". The GA version of OS/2 2.0 did not provide a toolkit to support these 16-bit API functions, so we took the necessary piece from the 16-bit toolkit. Compare to 32-bit DosQuerySysInfo.

The steps to use a 16-bit API from 32-bit OS/2 2.1 are: (1) put in a function prototype with either linkage keywords or the #pragma linkage; (2) add any other structure definitions required; and (3) deal with the return information. It is as simple as that! This is supported by some "magic" in the C runtime and by OS2286.LIB.

```
link386 /DE /NOI $*.obj , , ,os2386.lib os2286.lib, $*.def;
```


EDCThunkProlog, DosFlatToSel, DosSelToFlat

```
main      PUBLIC main
PROC
PUSH     04H
PUSH     02060H          <- 16 bit stack size 8K+
CALL     ___EDCThunkProlog  <- insure stack not cross 64K
PUSH     EBP
MOV      EBP,ESP
SUB      ESP,0cH

;***** 23   rc=DosGetInfoSeg(&selG,&selL);

SUB      ESP,038H
LEA     EAX,[EBP-04H];   selG
CALL    _DosFlatToSel    <- thunk 32 to 16

PUSH    EAX
LEA     EAX,[EBP-02H];   selL
CALL    _DosFlatToSel    <- thunk 32 to 16
.
.   code eliminated
.

;***** 31   pginfoseg = MAKEP (selG,0);

XOR     EAX,EAX
MOV     AX,[EBP-04H];   selG
SAL     EAX,010H
CALL    _DosSelToFlat    <- thunk 16 to 32
MOV     [EBP-08H],EAX;   pginfoseg
```

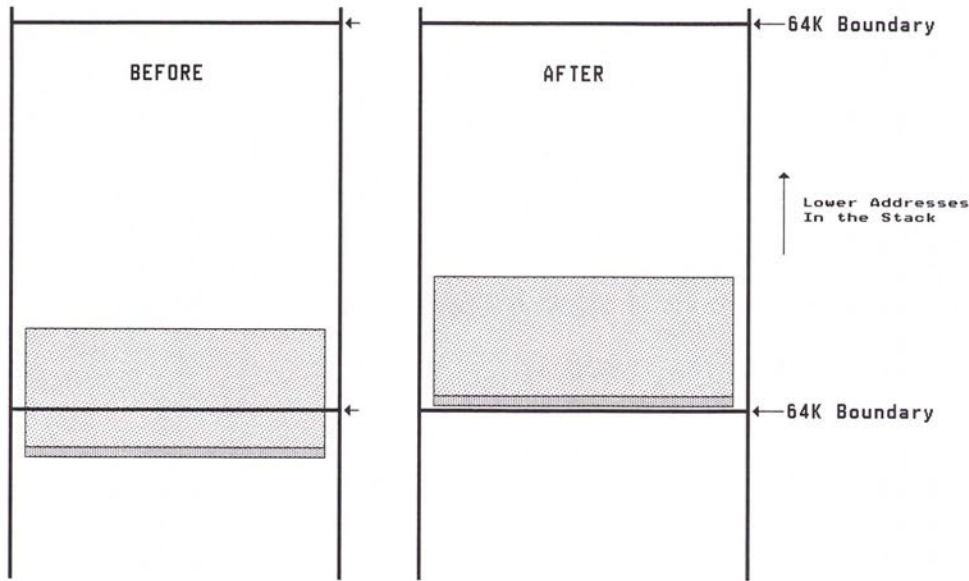
The machine code above generated by the compiler shows the reservation of 8 Kbytes (2000H) of Stack for the 16 bit API (PUSH 02060H) using ___EDCThunkProlog. We'll look at that on the next page. The code also shows the "modern" way of thinking addresses -- using _DosFlatToSel and _DosSelToFlat.

DosFlatToSel and DosSelToFlat are used for address conversion (thunking) because there are some exceptions to the conversion algorithm due to code segment packing in some .EXE files. If you look at the machine code of this program using IPMD or Kernel Debug, you will see that the calls to _DosFlatToSel are *fixup* records at load time and thus do not require extra machine cycles!!

Note that code generated for line #31 shows the MAKEP macro as three instructions (since the offset was 0), and then shows the call to _DosSelToFlat since we are thunking across the assignment operator.

I am not showing you the actual trace of the call from 32 to 16. There is a routine called _EDC3216 which does the actual thunk. It is inside of the C runtime.

EDCThunkProlog and #pragma stack16 ()



The `#pragma stack16(8192)` overrides the default stack size for the 16 bit call (which is 4K). This tells `__EDCThunkProlog` that `2000H` + some overhead, or 8K, of stack is needed by the 16 bit function. If the current stack value in ESP, - 4 for parameters, and - `0x2060` for the 16 bit function, would cross a 64K boundary, then `__EDCThunkProlog` will move ESP to the far (low) side of the 64K boundary and thus the stack for the 16 bit function will not cross a 64K boundary.

```
PUSH    04H
PUSH    02060H
CALL    __EDCThunkProlog
```

<- 16 bit stack size 8K +
<- insure stack not cross 64K

A 32-bit program may have lots of stack allocated, but since the 16-bit API can only use stack within 64K boundaries, the 16-bit API might run out of stack if ESP is approaching a `0x10000` boundary.

In the illustration the 32-bit stack (remember ESP decrements as you use stack) is moving lower (vertically up) and about to hit a 64K boundary. That is, if the 16-bit API to be called uses 8K of stack, *its* stack will cross the line. `__EDCThunkProlog` uses the desired target stack size and the current value in ESP to see whether or not the 16-bit API would have a problem. If there is not sufficient "running room" in the stack, `__EDCThunkProlog` moves ESP lower. When the 16-bit call returns, the stack is put back.

The heavy shaded area represents the stack frame (parameters) for the target API, and the lightly shaded area represents the stack required by the target API.

Calling a 16 Bit DLL from 16 Bit Code

```
#define INCL_BASE // This is the MAIN program
#include <os2.h>
#include <stdio.h> // for printf

void wait_key (char *ascii, char *scan);

void main ( )
{
    char chAscii, chScan;
    wait_key ( &chAscii, &chScan );
    printf ("\n The Character Code is %2X \n The Scan Code is %2X \n",
           chAscii, chScan);
    DosExit ( EXIT_PROCESS, NO_ERROR);
}

#define INCL_BASE // This is the 16-Bit DLL
#include <os2.h>
#include <stdio.h>

void wait_key ( char *ascii, char *scan )
{
    KBDKEYINFO Kbd;
    HKBD hKbd = 0;
    USHORT rc;
    printf ("\n DLL Entered \n");
    rc = KbdCharIn ( &Kbd, IO_WAIT, hKbd );
    if (rc) terminate ("Error in wait_key\n", rc);

    *scan = Kbd.chScan;
    *ascii = Kbd.chChar;
}
```

This program appears in \OS22LABS\SAMPLES\KEY16DLL. It was written and compiled using 16-Bit OS/2. The 16-Bit EXE and 16-Bit DLL are in the directory. Run as MAIN.EXE and hit any character on the keyboard to test.

The program calls a DLL which using 16-bit API KbdCharIn to read a character. It returns the character code and the scan code. On the next page, we will keep the same DLL but use a 32-bit main, thus illustrating what we need to do in a 32-bit main to call a 16-bit DLL.

Calling a 16 Bit DLL from 32 Bit Code

```
#define INCL_BASE                                \\ 32-bit MAIN
#include <stdio.h>
#include <os2.h>

#pragma stack16(8192)                            \\ specify 8K of stack
#pragma linkage (wait_key, far16 cdecl)          \\ this pragma does it!

// This 16 bit function resides in UTIL.DLL

void wait_key (char *ascii, char *scan);

void main ( )
{
    char chAscii, chScan;
    wait_key ( &chAscii, &chScan );
    printf ("\n The Character Code is %2X \n The Scan Code is %2X \n",
            chAscii, chScan);
    DosExit ( EXIT_PROCESS, NO_ERROR);
}
```

main.exe: <- THE MAKEFILE

```
main.obj:      *.c *.def makefile
icc /c /Ss /Fa /Ti /O- /Gd- *.c > *.err
type main.err
```

```
main.exe: *.obj *.def
link386 /DE /NOI *.obj,,,util.lib os2386,*;
erase *.map
```

This program appears on the advanced class diskette as \OS22LABS\SAMPLES\KEY32DLL. It was written and compiled using 32-Bit OS/2. The 32-Bit EXE and 16-Bit DLL are in the directory. Run as MAIN.EXE and hit any character on the keyboard to test. UTIL.DLL and UTIL.LIB were copied from ..\KEY16DLL.

To save space on the class diskette, files UTIL.LIB and UTIL.DLL need to be copied from KEY16DLL to KEY32DLL. Do this by running README.CMD in \OS22LABS\SAMPLES\KEY32DLL.

As you can see, it was effortless to call the 16-bit DLL. We just put in the `#pragma linkage` specifying `far16 cdecl`. We used `cdecl` because the code inside of the DLL uses `cdecl` (not `pascal`) linkage. UTIL.LIB is the import library for the DLL.

16 Bit Callback to 16 Bit Main

```
#define INCL_BASE    /* ---- 16 BIT Main ---- */
... etc.

typedef int  (cdecl far *PCFN)();
void wait_key (PCFN pfunction);
int _loadds printit (CHAR chAscii, CHAR chScan);

void main ( )
{
    wait_key ((PCFN) printit);
    DosExit ( EXIT_PROCESS, NO_ERROR);
}

int cdecl far _loadds printit (CHAR chAscii, CHAR chScan)
{
    printf ("\n The Character Code is %2X \n The Scan Code is %2X \n",
        chAscii, chScan);
    return(0);
}

#define INCL_BASE    /* ---- 16 BIT DLL ---- */
... etc.

void wait_key (PVOID x)
{
    void ( far *pfnPrintit) (CHAR chAscii, CHAR chScan);
    KBDKEYINFO Kbd;           // keyboard data structure
    USHORT rc;               // result code
    printf ("\n DLL Entered \n");
    pfnPrintit = x;
    rc = KbdCharIn ( &Kbd, IO_WAIT, 0 );
    if (rc) printf ("KbdCharIn Error \n");
    pfnPrintit (Kbd.chChar, Kbd.chScan);
}
}
```

This program appears in \OS22LABS\SAMPLES\CALLBK16. It was written and compiled using 16-Bit OS/2. The 16-Bit EXE and 16-Bit DLL are in the directory. Run as MAIN.EXE and hit any character on the keyboard to test.

The program passes the address of procedure **printit** to the DLL. The DLL does an indirect call through a pointer with parameters to do the callback. This is the standard technique we use in lab. The **_loadds** was put in because the compiler switches to the DLL's data segment when the DLL is called, and we need it to switch back to main's data segment when we return to the main. The 16 bit qualifier **_loadds** also emit code to restore the caller's DS so the DLL would continue to run after the return from **printit**.

16 Bit Callback to 32 Bit Main

```
#define INCL_BASE
#include <stdio.h>
#include <os2.h>

void _Far16 _Cdecl wait_key (void (* _Far16 _Cdecl)(char, char));
void _Far16 _Cdecl printit (CHAR chAscii, CHAR chScan);

void main ( )
{
    wait_key (printit);
    DosExit ( EXIT_PROCESS, NO_ERROR);
}

void _Far16 _Cdecl printit (CHAR chAscii, CHAR chScan)
{
    printf ("\n The Character Code is %2X \n The Scan Code is %2X \n",
        chAscii, chScan);
    return;
}
```

To save space on the class diskette, files UTIL.LIB and UTIL.DLL need to be copied from KEY16DLL to CALLBK16. Do this by running README.CMD in \OS22LABS\SAMPLES\CALLBK32.

This program appears in \OS22LABS\SAMPLES\CALLBK32. It was written and compiled using 32-Bit OS/2. The 32-Bit EXE and 16-Bit DLL are in the directory. Run as MAIN.EXE and hit any character on the keyboard to test. UTIL.DLL and UTIL.LIB were copied from ..\CALLBK16.

Note the linkage words **_Far16 _Cdecl** are inline in the function prototypes and the function itself. This is better style than using #pragma linkage. Both work equally well.

The program shows that 16-bit linkage actually means two things: it means that when calling a 16-bit routine from 32-bit code, pass the parameters to the 16-bit target in the manner it expects. It also means that if a function inside of a 32-bit application is declared to be a 16-bit function, then it *picks up* its parameters from the stack and/or registers in the 16-bit manner. Thus routine printit has been emitted as 32-bit code with a 16-bit front end!

Using the callback there is some limit to the amount of data which may be passed in the parameter list (perhaps 32 words). Do not pass structures by value using a callback, and carefully check the documentation.

LDT allocation in 16-Bit Program

```
#define INCL_BASE    /* ---- 16 BIT Main ---- */
... etc.

void PassTo16 (SEL sel);

void main ( )
{
    USHORT rc;
    SEL    sel;

    rc = DosAllocSeg (200, &sel, SEG_NONSHARED);
    if (rc) printf ("MAIN: DosAllocSeg Failure\n");
    printf ("Private Segment at %4X \n", sel);
    PassTo16 (sel);
    DosExit (EXIT_PROCESS, 0);
}

#define INCL_BASE    /* ---- 16 BIT DLL ---- */
... etc.

extern USHORT APIENTRY usSegmentSize (SEL sel);

void PassTo16 (SEL sel)
{
    USHORT rc;
    SEL    sel1;
    printf ("\n DLL Entered \n");

    printf ("Private Segment at %4X \n", sel);
    printf ("Segment Limit is %d \n", usSegmentSize (sel));

    rc = DosAllocSeg (200, &sel1, SEG_NONSHARED);
    if (rc) printf ("UTIL: DosAllocSeg Failure\n");

    printf ("\nPrivate Segment at %4X \n", sel1);
    printf ("Segment Limit is %d \n", usSegmentSize (sel1));
    return;
}
```

This program appears in \OS22LABS\SAMPLES\LDT16DLL. It was written and compiled using 16-Bit OS/2. The 16-Bit EXE and 16-Bit DLL are in the directory. Run as MAIN.EXE.

The printout from this program appears in the left column of the next page. The 16-bit main allocates a 200 byte segment and passes the selector to the DLL. The 16-bit DLL also allocates a 200 byte segment. In both cases the LDT contains a limit of 199, which is what we expect in a 16-bit environment.

16/16 and 32/16 Printout

| 16 Bit Main and 16 Bit DLL | 32 Bit Main and 16 Bit DLL |
|--|---|
| Private Segment at 47 | Private Segment at 8F0000 |
| DLL Entered Private Segment at 47 Segment Limit is 199 | DLL Entered Private Segment at 8F Segment Limit is 4095 |
| Private Segment at 57 Segment Limit is 199 | Private Segment at 97 Segment Limit is 199 |

One interesting question arises: let's say that a 32-bit main allocates some memory using `DosAllocMem`, and then passes the selector of that memory object to a 16-bit DLL. Who builds the LDT? When is it built? Does, in fact, every 32-bit application have its own LDT in 32-bit OS/2.

The program on the next page and the printout above demonstrate that each 32-bit process *does* have its own LDT and a `DosAllocMem` of 200 bytes will actually create an LDT entry with a limit of 4095! However, when the 16-bit DLL allocates 200 bytes, it creates an LDT entry with a limit of 199.

Thus:

16 Bit Applications Running under 32-Bit OS/2 set the LIMIT field in the LDT to the exact value of the Segment Limit.

32 Bit applications use an LDT entry when they do a `DosAllocMem`. The limit of the LDT entry is the actual memory limit, here 4095.

If a 32-bit application passes a selector to a 16 bit DLL, the selector allocated by the 32-bit application has a minimum length of 4K, whereas a selector allocated by the DLL has a length equal to the allocation size.

LDT allocation in 32-Bit Program

```
#define INCL_BASE
#include <stdio.h>
#include <os2.h>

// #define SELECTOROF(ptr)  (((ULONG)(ptr))>>13)|7)  <- OS2DEF.H

// This 16 bit function resides in UTIL.DLL

void _Far16 _Cdecl PassTo16 (SEL sel);

#define ALLOCFLAGS PAG_READ | PAG_WRITE | PAG_COMMIT | OBJ_TILE

void main ( )
{
    APIRET rc;
    PCH pch32;          // linear address of our private array
    PCH _Seg16 pch16;  // 16:16 address of our private array

    rc = DosAllocMem ((PPVOID) &pch32, 200, ALLOCFLAGS);
    if (rc) printf ("MAIN: DosAllocMem Failure\n");
    pch16 = pch32;
    printf ("Private Segment at %8X \n", pch16);

    PassTo16 ((SEL) pch16 >> 16);
    DosExit ( EXIT_PROCESS, NO_ERROR);
}
```

This program appears in \OS22LABS\SAMPLES\LDT32DLL. It was written and compiled using 32-Bit OS/2. The 32-Bit EXE and 16-Bit DLL are in the directory. Run as MAIN.EXE. It uses the 16-bit DLL and import library from ..\LDT16DLL. The printout appears on the previous page.

In order to save space on the class diskette, the 16-bit import library and DLL are not in this directory. Copy them in by running README.CMD.

We did not use the pre-defined macro SELECTOROF which calculates the selector from a 16:16 pointer. Instead, we simply shifted pch16 to the right 16 bits.

This program demonstrates that 32-bit programs running under OS/2 2.0 have a 16-bit LDT associated with them. Thus, when we allocated 200 bytes of memory from the 32-bit code and passed the selector to the 16-bit DLL, the 16-bit DLL reported (in the second column on the previous page) that we had allocated 4K bytes at that selector:offset address.

This explains while OBJ_TILE is the default in OS/2 2.1: every DosAllocMem and DosAllocSharedMem uses at least one 64K LDT entry in the 32-bit processes LDT!